The University of Jordan

Authorization Form

I, Somaya Saber Ahmad Ismail, authorize the University of Jordan to supply copies of my Thesis/ Dissertation to libraries or establishments or individuals on request, according to the University of Jordan regulations.

Signature:

Date: 10/5/2011

الجامعة الأردنيــــــة
كلية الدراسات العليا

التاريـــخ: / /

نموذج رقم (١٨)
اقرار والتزام بالمعايير الأخلاقية والأمانة العلمية
وقوانين الجامعة الأردنية وأنظمتها وتعليماتها
لطلبة الماجستير

أنا الطالب: سميرة جابر أحمد كامل الرقم الجامعي: (8062160)

تخصص: نظم معلومات حاسوبية الكليــــــة: تكنولوجيا المعلومات

عنوان الرسالة: ....
Using Ant Colony optimization for Regression Test Case prioritization
....

اعلن بأنني قد التزمت بقوانين الجامعة الأردنية وأنظمتها وتعليماتها وقراراتها السارية المفعول المتعلقة باعداد رسائل الماجستير عندما قمت شخصيا" باعداد رسالتي وذلك بما ينسجم مع الأمانة العلمية وكافة المعايير الأخلاقية المتعارف عليها في كتابة الرسائل العلمية. كما أنني أعلن بأن رسالتي هذه غير منقولة أو مستلة من رسائل أو كتب أو أبحاث أو أي منشورات علمية تم نشرها أو تخزينها في أي وسيلة اعلامية، وتأسيسا" على ما تقدم فانني أتحمل المسؤولية بأنواعها كافة فيما لو تبين غير ذلك بما فيه حق مجلس العمداء في الجامعة الأردنية بالغاء قرار منحي الدرجة العلمية التي حصلت عليها وسحب شهادة التخرج مني بعد صدورها دون أن يكون لي أي حق في التظلم أو الاعتراض أو الطعن بأي صورة كانت في القرار الصادر عن مجلس العمداء بهذا الصدد.

توقيع الطالب: .... التاريخ: ١٠ / ٥ / ٢٠١١

USING ANT COLONY OPTIMIZATION FOR REGRESSION TEST CASE PRIORITIZATION

By

Somaya Saber Ahmad Ismail

Supervisor

Dr. Khalil El-Hindi

This thesis was submitted in partial Fulfillment of the Requirements for the Master's Degree of Computer Information System

Faculty of Graduate Studies

The University of Jordan

May, 2011

COMMITTEE DECISION

This Thesis/Dissertation (Using Ant Colony Optimization for Regression Test Case Prioritization) was Successfully Defended and Approved on 11/4/2011

Examination Committee	Signature
Dr. Khalil M. El-Hidi (Supervisor) Assoc. Prof. of Artificial Intelligence	
Dr. Ammar M. Huniti (Member) Assit. Prof. of Adaptive Hypermedia	
Dr. Mohammad Alshraideh (Member) Assit. Prof. of Software Testing Using Evolutionary Algorithms. (Member)	
Dr. Bayan Abo Shawar (Member) Assit. Prof. of Natural Language Processing (Open Arab University)	

DEDICATION

To the faithful wisdom and unmistaken foresight

Who shed love generously to all my family

Who stays close to me as stepping forward in the future

To whom Colored my life with love and beauty

And gave it a great meaning

Who spared no effort in help and support

My ever beloved parents

To those who encouraged me

through the duration of this thesis

And stood all the time with me

To whom we grow together learn and face together,

hand in hand the challenges life sends

My brothers and Friends

To all, I dedicate this work

ACKNOWLEDGMENT

This work would not have been completed in this way without the expertise, encouragement, patience, faith, and dedication of my supervisor Dr. Khalil El Hindi.

I would like to take this opportunity to thank my Supervisors for guiding me intellectually and spiritually during my work and for providing fruitful suggestions and sharp comments for the improvement of this thesis.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACO	Ant Colony Optimization
AS	Ant System
GA	Genetic Algorithms
APBC	Average Percentage Block Coverage
APDC	Average Percentage Decision Coverage
APSC	Average Percentage Statements Coverage

USING ANT COLONY OPTIMIZATION FOR RREGRESSION TEST CASE PRIORITIZATION

By

Somaya Saber Ahmed Ismail

Supervisor

Dr. Khalil El-Hindi

ABSTRACT

Software testing is a very important phase in software life cycle. It is aimed to evaluate program or system that it meets the business and technical requirements and works as expected. The importance of software testing is to provide stakeholders with information about the quality of the product or service under test. Regression testing is one of black box testing methods that do not consider the internal system design. Regression testing is performed frequently to ensure the validity of the modified software. Also, Regression testing is used to ensure that modifications are correct and have not adversely affected the unchanged portions of the software. In traditional regression testing, the same tests (the regression tests) are reused. The test suites that are already available from earlier versions of software can be expensive to execute completely.

Many approaches have been explored to minimize the cost and efforts of re execution regression test suites. Selecting, minimizing, and test case prioritizing are approaches to minimize regression testing cost. Regression test case prioritization technique schedule test cases located in test suites in order to achieve code coverage or fault detection as soon as possible. Regression test case prioritization is considered as NP-hard problem; which means for this class of problems no algorithm of polynomial complexity could be found. In this work, a biologically inspired heuristic (Ant Colony) is used to solve such problem. The forging behavior of real ants modeled algorithmically for solving some problems that require optimal solution. The proposed Ant Colony Optimization technique used to prioritize test cases is Ant System (AS) which looks into test suites to find the optimal order of them to maximize code coverage.

In this work, the proposed algorithm was applied on four programs assembled by researchers at Siemens Corporate Research for experiments and a test pool containing possible test cases for each program. Code coverage is used as a

surrogate measure and three metrics are evaluated; Average Percentage Block Coverage (APBC), Average Percentage Decision Coverage (APDC), and Average Percentage Statement Coverage (APSC). The solutions obtained in this thesis by this heuristic algorithm are compared with Genetic Algorithm. Our results show that ACO and GA outperforms each other on some bench mark test-case prioritization problems and none of them can substitute the other. Therefore, both of them may be applied on a given problem and the better test-case order might be used.

Chapter (1): Introduction

Regression testing is a frequently applied but expensive maintenance process that aims to (re)verify modified software. Onoma et al. (1998) reported that the secret in delivering quality software is good regression testing. Many approaches for improving the regression testing processes have been investigated. Test case prioritization is one of these approaches, which orders test cases so that the test cases with highest priority, according to some criterion are executed first.

Testers might wish to schedule test cases in order to achieve code coverage at the fastest rate possible in the initial phase of regression testing to reach 100 percent coverage soonest or to ensure that the maximum possible coverage is achieved by some predetermined cut-off point.

Ant Colony Optimization (ACO) is one of a metaheuristic algorithms that can find optimal solution for combinatorial optimization problems, , ranging from quadratic assignment to fold protein or routing vehicles and a lot of derived methods have been adapted to dynamic problems in real variables, stochastic problems, multi-targets and parallel implementations by constructive shortest path.

Regression test case prioritization aims to find the best order that reveals faults as soon as possible, thus finding optimal execution ordering for test cases located in testing pool is NP-hard problem. Finding near optimal order for this type of problem needs heuristic algorithm, thus ACO is proposed in this work to solve such problem.

1.1 Problem Description

Software testing is a very important step in software development life cycle; it is used to validate the correctness, the completeness, security, and quality of the developed computer software. Software testing includes the process of executing a program or application with the intent of finding errors, and to detect the difference between existing and the required functions and behavior.

A very important issue is that software testing starts once the development process starts; this means that it is needed to be done through the whole development process. The earlier a defect is found the cheaper it is to fix (Megen and Meyerhoff, 1995). This process leads the software engineers often to save the test suites they develop for their software so that they can reuse them in the form of the regression testing.

Regression testing is a frequently executed maintenance process used to revalidate modified software (Ma, et al., 2005). Whenever software is modified or evolved, new test cases are added to the test suite to test the modifications or system requirements, increasing the size ,cost ,effort ,and time of regression testing .For example , one of industrial collaborators reports that for one of its products of about 20,000 lines of code , the entire test suite requires seven weeks to run (Rothermel ,et al.,2001). For these reasons, researchers have considered various techniques for reducing the cost of regression testing, including regression test selection, test suite prioritization and minimization techniques.

Selection techniques reduce the cost of regression testing by selecting an appropriate subset of the existing test suite, based on information about the program, modified version, and test suite. Test suite minimization techniques

reduce costs by reducing the size of test suite to a minimal subset that maintains equivalent coverage of the original test suite with respect to particular test adequacy criterion (Rothermel, et al., 2001).

Test case Reduction method address the problem of regression testing by using information about the program ant test suite to permanently remove redundant test cases from test suite, rendering later reuse of test suite more efficient. Test cases Reduction reduce the size of the test suite which reduce the costs of executing , validating, and managing test suites over future releases of the software.

Test cases prioritization techniques involve scheduling the test cases in an order to improve the performance and effectiveness of regression testing so that the most beneficial cases are executed first. In the context of regression testing , prioritization techniques can take advantages of information gathered about the previous execution of test cases to obtain test case ordering .Test cases prioritization techniques do not discard test cases, for this reason , they can avoid the drawbacks that can occur when regression test selection and test suite minimization are executed (Rothermel , et al ., 2001).

1.2 Importance of the Problem

Establishing a policy for regular regression testing is the key to achieving successful, reliable, and predictable software development projects. Regression testing provides the only reliable means to verify that code base changes and additions do not introduce new errors in the code and functions. This can have

great impact in controlling product release delays, budget overruns, and the prospect of software errors slipping into released/deployed products. Yet regression testing is not widely practiced, because of time, effort, and cost needed. Development organizations often give up on regression testing because they find it complicated and difficult to maintain. But the major reason for failure with regression testing is the absence of a well-defined and implemented policy and an organizational commitment to that policy.

Optimizing the time and cost of the regression testing without compromising the fault exposing capability is always challenging for the testing team. Testing team always face constraints like lack of resources, squeezed testing schedule, changing and ambiguous requirements, which affects and reduces the effectiveness of regression testing.

Due to squeezed test schedule, testing team may not able to execute all test cases from the selected regression suite. Sequencing of test cases based on some criteria helps testing team to achieve the goals while reducing testing cycles. These efforts motivated researchers to propose several techniques and methods to find good schedules for regression test cases.

1.3 Main Contribution

ACO is classified as one branch of Artificial Intelligence known as swarm intelligence. ACO is used to solve many NP-combinatorial problems. In this thesis, the use of ACO as a new technique to prioritize test cases for regression testing is introduced. ACO uses a number of ants to interact with each other through the problem environment as indirect interaction. For this thesis ants used the un-prioritized test suites as an environment to look up the optimal order of test

cases to achieve code coverage as soon as possible. This thesis work aims to discover the effectiveness of the proposed algorithm which is not deployed in the literature yet. The results of our proposed algorithm are compared with the results of Genetic Algorithm. The experiments results shows that the proposed Ant System algorithm has good results compared to those results of Genetic Algorithm as will discussed in detail in chapter 3.

1.4 Organization of the Thesis

In order to follow the proposed study, it is important to clarify the organization of this thesis. This thesis is organized in the following chapters:

1. Chapter 2 literature review: This chapter attempts to clarify: the concept of software testing; regression testing; regression test case prioritization. Also a short brief discussion of metaheuristic concept; and an overview of Ant Colony Optimization (ACO) that will be used in this thesis. Finally, related work for our thesis is given.

2. Chapter 3 Ant Colony Optimization to prioritize Regression test case**:** This chapter aims to propose a new algorithm based on Ant System with new parameters and expertise it. Also Genetic Algorithm is tested for the same problem, and some statistical analysis is carried for the results in order to compare the results of algorithms.

3. Chapter 4 proposed Conclusion and Future Work: This chapter draws conclusion and presents points for future work.

Chapter 2: Literature Review

This chapter attempts to clarify: the concepts of Software Testing; Regression Testing; Regression test case prioritization, and some basic biological notions that inspired computer scientists to search of new algorithms to resolve optimization problems, and ACO. Also this chapter attempts to introduce several summaries of some previous efforts addressing similar problem. A reader, who is familiar with any of the presented section, may skip it to save her/his precious time.

2.1 Software testing

Computers and software are used widely in critical applications, so the outcome of a bug can be severe. Bugs in critical systems have caused airplane crashes, allowed space shuttle missions to go awry, halted trading on the stock market, and worse. In a computerized embedded world, the quality and reliability of software is a matter of life and death. So software testing is a very important step in software development life cycle; it is used to validate the correctness, the completeness, security, and quality of the developed computer software (Megen and Meyerhoff, 1995).

In 1991, Hetzel gave the definition that "Testing is planning, designing, building, maintaining and executing tests and test environments." A year before this, (Beizer, 1990) gave four stages of thinking about testing: (i) to make software work, (ii) to break the software, (iii) to reduce risk, and (iv) a state of mind (luo, 2009).

A very important issue is that software testing starts once the development process starts; this means that it is needed to be done through the whole development process and the cost of maintenance is increasing. For example, in the 1990s, the estimated

expenditure exceeds 70%of total software costs (Rothermel, et al., 1997). Common estimates indicate that a problem that goes undetected and unfixed until a program is actually in operation can be 40 – 100 times more expensive to resolve than resolving the problem early in the development cycle, hence , testing has a high priority of any software development effort. As shown in Figure 1. (luo, 2009).

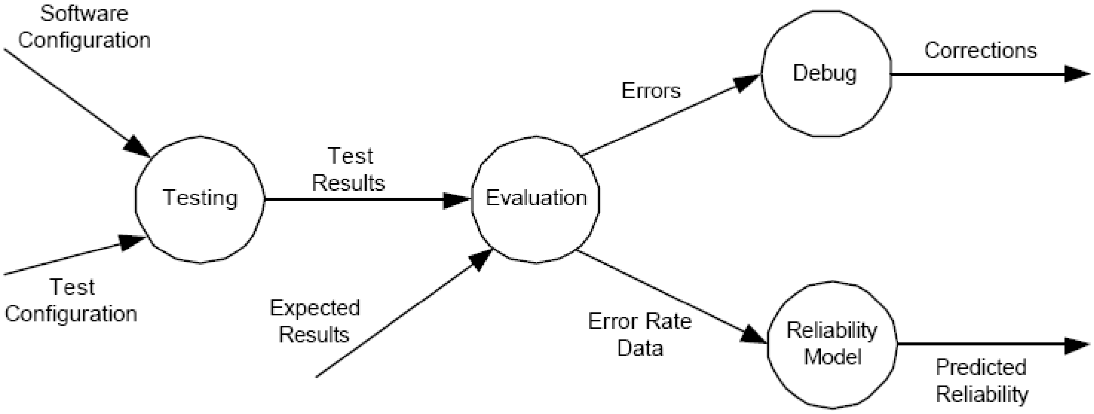


Figure 1: Software Testing Information Flow.

Testing can serve as metrics; testers can make claims based on interpretations of the testing results. We can also compare the quality among different products under the same specification, based on results from the same test. Good testing requires a tester's creativity, experience and intuition, together with proper techniques.

Testing divided into two types: black box testing and white box testing. The difference between them that black box testing tests the functionality of an application or software without knowledge of internal structure, on other hand, white box testing is called glass testing, which tester must know the internal structure, and must have programming skills.

Software testing can be categorized as follows: Component testing, Integration testing, System testing, and Acceptance testing.

Component testing also called unit testing is the process of testing the smallest piece of testing software by isolating it from the remainder of the code, and determine whether it behaves exactly it is expect. The aim of component testing stage is to discover defects by testing individual components. These components may be functions, objects or reusable components. Each component is tested separately before integrating them into modules to test the interfaces between modules. Component testing has proven its value in that a large percentage of defects are identified during its use.

Integration testing is the process of system integration involves building a system from its components and testing the resultant system for problems that arise from component interactions. Integration testing checks that components actually work together are called correctly and transfer the right data at the right time across their interfaces.

System testing involves integration two or more components that implement system functions or feature and then testing this integrated system. The main two phases of system testing are integration testing and release testing. In release testing where a version of the system that could be released to users is tested.

Acceptance testing involves running a suite of test cases on completed system. Each test case designed for proper operating condition. The results obtained are compared with user expected results.

Testing does not end by the conclusion of acceptance testing. Software has to be maintained to fix problems which show up during use and to accommodate new requirements. Software tests have to be repeated, modified and extended. The effort to

revise and repeat tests consequently forms a major part of the overall cost of developing and maintaining software. The concept Regression testing is used to refer to the repetition of earlier successful tests in order to make sure that changes to the software have not introduced new errors (Beni, et al., 1989).

Testing must be designed in order to get good results. Good design consists of four phases; test strategy, test planning, test case design, and test procedure design.

2.2 Regression Testing

Regression testing is a frequently executed maintenance process used to revalidate modified software (Ma, et al., 2005). As new requirements and changes have been made to software, the need to rerun test cases from test suites and new test cases added to the test suite to test the modifications is important to build confidence that the modified one is working probably and have no unintended side-effects.

The formal definition of regression testing is expressed as follows:

Let P be a program, let P' be a modified version of P, and let T be a test suite developed for P. Regression testing is concerned with validating P'.

Regression testing is an integral part of software development method. In this method, design documents are replaced by extensive, repeatable, and automated testing of the entire software package at every stage in the software development cycle.

As shown in Figure2. (Agrawal, et al., 1993).

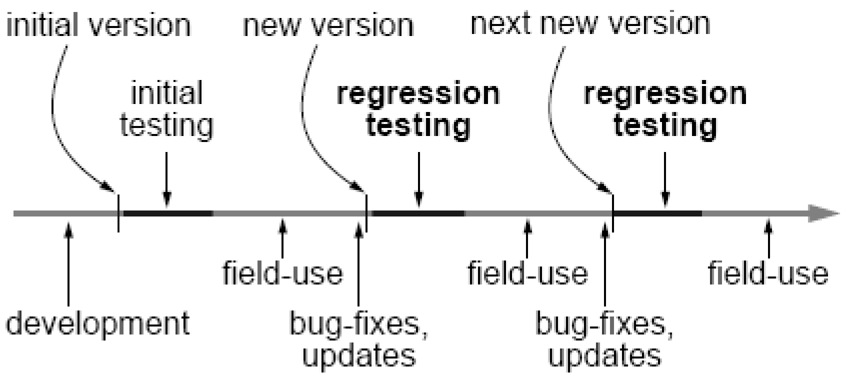


Figure 2: Regression Testing.

Regression testing is expensive but important process that increases software reliability. Onoma et al. (1998) reported that the secret in delivering quality software is good regression testing. Development organizations often give up on regression testing because they find it complicated and difficult to maintain and it needs more time, effort, and cost. But the major reasons for failure with regression testing are lack of resources, squeezed testing schedule, changing and ambiguous requirement, which affects and reduces the effectiveness of regression testing.

For these reasons, researchers suggest many methodologies to minimize the cost of Regression testing. The simplest Regression testing method is retesting all. Test engineers may simply reuse all test cases in the test suite and rerun them on the modified program. The main disadvantage of this approach is it can be prohibitively expensive, because rerunning all test cases in the test suite require a huge an unacceptable amount of time and human effort (Rothermel, et al., 2004).

On the other hand, an alternative method suggested, Regression Test Selection (RTS). RTS reruns only a subset of the initial test suite using information about the program and the modified one. This method is not very well, since Regression Test

Selection techniques (RTS) can have substantial costs, and can discard test cases that could reveal faults, possibly reducing fault detection effectiveness. In general, for an RTS technique to be cost-effective, the cost of performing the selection plus the cost of rerunning the selected subset of test cases must be less than the overall cost of rerunning the complete test suite (Rothermel, et al., 2004).

Test case Reduction method address the problem of regression testing by using information about the program and test suite to permanently remove redundant test cases from test suite, exposé later reuse of test suite more efficient. Test cases Reduction reduce the size of the test suite which reduce the costs of executing , validating, and managing test suites over future releases of the software.

On the other hand, regression test prioritization method is used to select all test cases with a priority for the most effective ones. This method will not eliminate some test cases that occur in RTS but it takes a long time to run all test cases although they are ordered. Regression test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time will have been spent more beneficially than if test cases were not prioritized.

2.3 Regression Test Case Prioritization

Optimizing the time and cost of the regression testing without compromising the fault exposing capability is always challenging for the testing team. Test case prioritization techniques involve scheduling over test cases that are located in test suite in an order that improves the performance and effectiveness of regression testing so that the most beneficial cases are executed first.

Test case prioritization problem was first introduced by Wong et al. (1997) as a flexible method of software regression testing. Their technique first selects test cases based on modified code coverage and then prioritizes them. Research in this context has been followed by Rothermel, Elabaum and other researchers, and various techniques for test cases prioritization have been introduced.

Test case prioritization addresses a wide variety of objectives:

- Testers need to schedule test cases in order to achieve code coverage at the fastest rate possible in the initial phase of regression testing to reach 100 percent coverage soonest or to ensure that the maximum possible coverage is achieved.
- Tester need to increase the likelihood of revealing faults related to specific code changes earlier in the testing process.
- Tester need to increase their confidence in the reliability of the system under test at faster rate.

Rothermel et al., (2001) define the test case prioritization problem as follows:

Given: *T, a test suite; PT the set of permutations of T; a function from PT to the real numbers.*

Problem: Find T'∈ PT such that

$$(\forall T)\ (T'' \in PT)(T'' \neq T')\ [f(T' \geq f(T'')]$$

In this definition, *PT* represents the set of all possible prioritization of *T*, and *f* is a function that applies to any such ordering, yield an award value for that ordering.

Test case prioritization is divided into two varieties: general test case prioritization and version-specific test case prioritization. For general test case prioritization, given program *P* and test suite *T*, test case prioritize the test cases in *T* with the intent of finding ordering of test cases that will be useful over a succession of subsequent modified version of *P*. general test case prioritization can be performed following the release of some version of the program.

While version-specific case prioritization, for given program *P* and test suite *T*, test case prioritization technique prioritize the test cases in T with the intended of finding an order that be useful on specific version *P'* of *P*. It is performed after a set of changes on *P* have been made.

The main difference between general test case prioritization and version –specific test case prioritization, that general testing does not use information about specific modified version *P*, whereas version-specific prioritization does.

For test case prioritization problem, given a function *f* that quantifies whether a test suite achieves statement coverage at the fastest rate possible, an efficient solution to the test case prioritization problem would provide an efficient solution to the knapsack problem, hence, test case prioritization problem considered to be intractable problem. Figure 3 shows an example of procedure and its related test cases.

For such a problem, test case prioritization techniques must be heuristic.

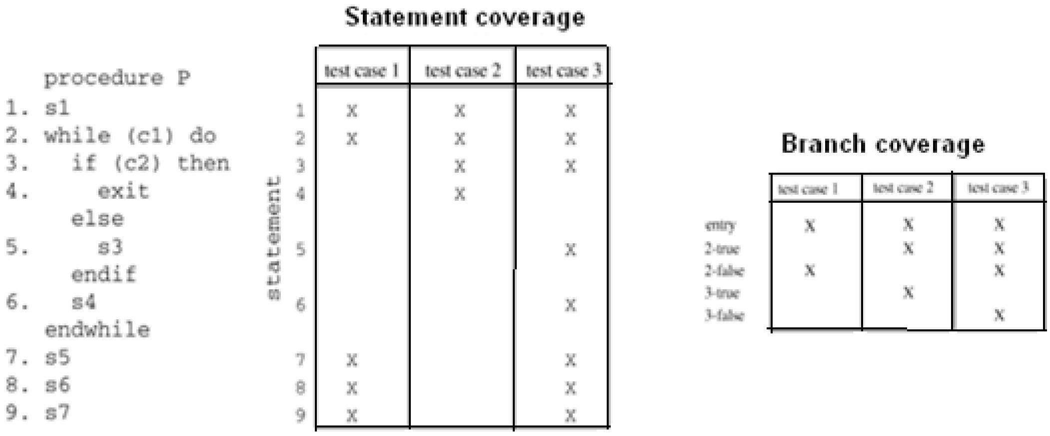


Figure 3: Example procedure P and related test cases, also statements and branch coverage.

Test case prioritization techniques divided into categories depending on the goal it achieved. Most of these techniques are code-based, depending on information relating test cases to coverage of code elements. Other techniques are non-coverage based such as techniques depends on fault-exposing potential (FEP) (Rothermel, et al., 2001), history-based test prioritization (Kim and Porter, 2002), (Park, et al.,2008), model-based prioritization (Korel, et al.,2007) , and the incorporation of varying test costs and fault severities into test case prioritization (Elbaum, et al., 2001), (Elbaum, et al., 2002). Alternatively, test case prioritization technique might prioritize test cases in terms of their increasing cost-per-coverage of code components, or in terms of their increasing cost-per-coverage of features listed in a requirements specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can meet the goal than would an ad hoc or random ordering of test cases. (Rothermel, et al., 2001).

An example to explain Regression Test Case Prioritization, consider a program with a test suite containing four test cases, A, B, C, D, and E, such that the program has ten

blocks covered by these test cases, as shown in Table 1. Test case prioritization aims to achieve full block coverage as soon as possible.

Table 1. Test Suite and Block Coverage

Test Case	block									
	1	2	3	4	5	6	7	8	9	10
A	X		X							
B	X	X	X		X					
C	X	X	X	X	X	X				
D		X		X						
E							X	X	X	X

Fitness metrics studied in this work are based upon three coverage metrics which are:

1. APBC (Average Percentage Block Coverage): this measures the rate at which a prioritized test suite covers the blocks.
2. APDC (Average Percentage Decision Coverage): this measures the rate at which a prioritized test suite covers the decisions.
3. APSC (Average Percentage Statement Coverage): this measures the rate at which a prioritized test suite covers the statements.

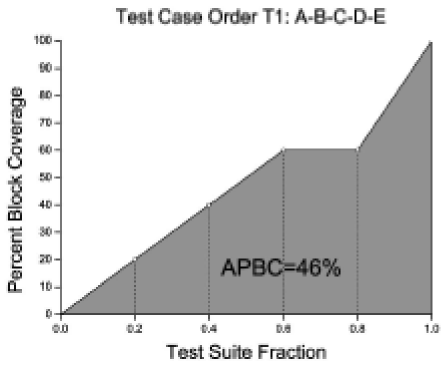
Consider the APBC metric as an example and a test suite *T* containing n test cases that covers a set B of *m* blocks. Let TB_i be the first test case in the order *T*' of *T* that covers block *i*. The APBC for order *T'* is given by the equation

$$APBC = 1 - \frac{TB_1 + TB_2 + \cdots + TB_m}{nm} + \frac{1}{2n} \qquad (1)$$

APBC measures the weighted average of the percentage of block coverage over the life of the suite. APBC values range from 0 to 100; higher numbers imply faster (better) coverage rates.

These metrics are used in this work as fitness function for AS and GA algorithms.

Considering the previous example, Table 1, two orders of these test cases are supposed, order T1: A-B-C-D-E, and T2:C-E-B-A-D. Figure 4 and figure 5 show the percentage of block coverage as a function of the fraction of the test suite used, for the two orders T1 and T2, respectively. The area under the curve represents the average of the percentage of block coverage over the life of the test suite. (Li, et al., 2007).



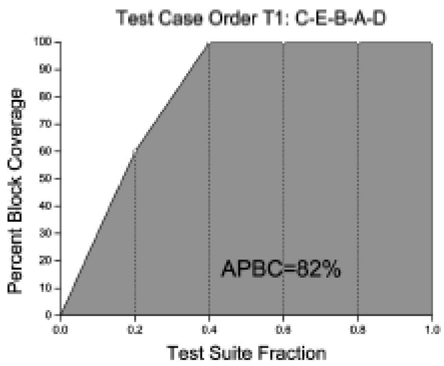


Figure 4. APBC for order T1. **Figure 5.APBC for order T2.**

This example shows that T2 will achieve code coverage faster than T1, since APBC value for T2 is higher. So good order for test cases, will improve the efficiency of Regression Testing.

2.4 Ant Colony Optimization Metaheuristic

In this section, we will briefly present some basic biological concepts used in computer science motivated researchers to find new algorithms to resolve optimization problems.

Also this section will describe the concepts of optimization problems, metaheuristic, and Ant Colony Optimization Algorithms with its variants. Finally, related techniques for Regression test case prioritization are mentioned.

2.4.1 Swarm Intelligence

Swarm Intelligence (SI) concept refers to the collective behavior of decentralized, self organized systems, natural or artificial agents (Beni and Wang, 1989). SI systems are typically made up of a population of simple agents interacting locally with one another and with their environment. These simple agents are self-organized control and cooperation; they form a distributed system of interacting agents with indirect interactions. Insects (ants, wasps and termites) that live in colonies are able to perform different sophisticated activities like foraging, corpse clustering, larval sorting, nest building, transport cooperation and dividing labor among individuals (Bonabeau, et al., 1999).

This behavior of social insects (finding food, building a nest, efficiently dividing labor among individuals, etc) inspired researchers in computer science and engineering to employ swarm intelligence in many fields such as solving combinatorial problems.

Swarm intelligence offers an alternative way of designing intelligent systems, in which autonomy, emergence and distributed functioning, replace control, preprogramming, and centralization.

Swarm Intelligent techniques used in multi applications. For example, the U. S military used SI for controlling unmanned vehicles. NASA investigated the use of SI for planetary mapping (Wikipedia, 2011). Lewis and George discussed the possibility of using SI into controlling nano robots within the body for the purpose of killing cancer tumors (Lewis and George, 1992).

2.4.2 Self-Organization in social insects

Self-organization was originally introduced in the context of physics and chemistry to describe how microscopic processes give rise to macroscopic structures in out-of-equilibrium systems. In biological systems self-organization concept is defined as " a process in which pattern at the global level of a system emerges solely from numerous interactions among the lower-level components of the system. Moreover, the rules specifying interactions among the systems components are executed using only local information, without reference to global pattern" (Camazine, et al.,2003).

Self –Organization is based in four ingredients (Bonabeau, et al., 1999):

1. Positive feedback (amplification): it is constituted by simple behavioral rules that promote the creation of structures, such as recruitment and reinforcement. Recruitment to a food source is a positive feedback that relies on trail laying and trail following in some ant species, or dances in bees.
2. Negative feedback: it is for counter-balance positive feedback and to stabilize the collective pattern. It can be form of saturation, or competition, such as food source exhaustion, crowding at the food source.
3. Amplification of fluctuations: such as random walks, random task-switching. Randomness is often crucial because it enables the discovery of new solutions, and

fluctuations can be seeds from which structures nucleate and grow. For example, ants foraging for food source may get lost because they follow wrong trail but they can explore another unexploited food source and recruit nestmates to this food source.

4. Multiple interactions: self-organized structure requires a minimal density of mutually tolerant individuals that insects able to make use of the results of their own activities as well as others activities. For example, ants can use pheromone deposited by others on the trail as well as individual pheromone.

2.4.3 Optimization Problems

Optimization is the process of finding optimal (minimum or maximum) output or result. The optimum can be achieved by finding the best combination of input parameters. The main difficulty in optimization problems is determining if a given minimum is the best (global) minimum or a suboptimal (local) minimum (Haput and Haput, 2004).

An adequate model for optimization problem can be described as follows:

A combinatorial optimization problem

A model P=(S,Ω,f) of a combinatorial optimization problem consists of:

1. a search space S defined over a finite set of discrete decision variables X_i, $i=1,\ldots,n$; which consists of set of components C.
2. a set Ω of constraints among the variables; and
3. an objective function $f : 2^C \rightarrow R^+$ to be minimized.

The generic variable X_i takes values in $D_i = v_i^1, \ldots, v_i^{|D_i|}$. A feasible solution $s \in S$ is a complete assignment of values to variables that satisfies all constraints in Ω. A solution $s^* \in S$ is called a global optimum if and only if: $f(s^*) \leq f(s)\ \forall s \in S$.

2.4.4 Ant Colony Optimization Algorithm

In the 20th century a French entomologist called Grasse (1959) discovered some of species of termites interacts with themselves and its environment in a significant stimuli which referred by them as stigmergy. He defined it as: "Stimulation of workers by the performance they have achieved." Stigmergy is indirect, non-symbolic communication mediated with environment with local information that only accessed by these insects. Deneubourg and Gross (1990) attempted to clarify this communication through the pheromone of ants which is a substance they lay on their way toward food by conducting an experiment called “double bridge experiment”; this experiment contained two paths with the same length from nest of Ants (colony) to food, they observed that one of paths has concentration of Ants which result of more substance of pheromone on this path. Deneubbourg and Gross (1990) also conducted another experiment which is variant of “double bridge experiment” by using two paths with different lengths, this experiment illustrated that the shortest path has converges of ants.

Ant Colony Optimization inspired from the foraging behavior of ant colonies, through their capable of finding the shortest path from a food source to the nest without using visual cues. Also, they are capable of adapting to changes in the environment, for example finding a new shortest path once the old one is no longer feasible due to a new obstacle, as shown in Figure 6.

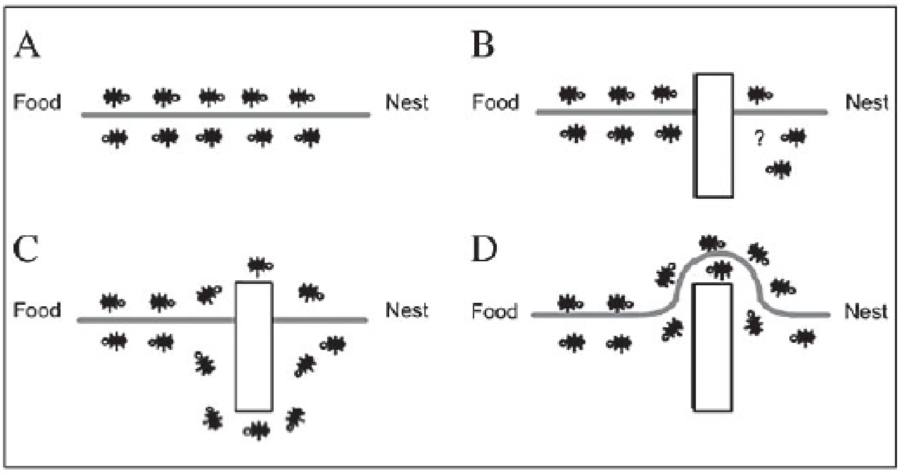


Figure 6: Ants' moving. (A) Real ants follow a path between nest and food source. (B) An obstacle appears on the path: Ants choose whether to turn left or right with equal probability. (C) Pheromone is deposited more quickly on the shorter path. (D) All ants have chosen the shorter path.

ACO is a paradigram that originally developed for combinatorial optimization problems as metaheuristic algorithm. It is general purpose framework which can be applied for optimization problems (Dorigo et al., Stutzle and Hoos, 2000; Dorigo and Gambrradella, 1997). Real ants deposit a chemical substance called pheromone on their path between nest and colony to be marked for favorable path for other ants to follow in the future. Ants work blindly with dynamic memory behavior and on parallel to search near optimal solution. The original Ant Colony Optimization algorithm was introduced in 1990s by Marco Dorigo and colleagues (Dorigo et al., 1991), and then numbers of other ACO algorithms were introduced. Table 2 list ant algorithms and their applications.

Table (2). Ant Algorithms and their applications.

Algorithm name	Developers	Year	Problem
Ant system	Dorigo et al.	1991	Traveling salesman problem
	Forsyth and Wren	1997	Bus driver scheduling
	Nahas and Nourelfath	2005	Reliability optimization of a series system
AS-QAP	Maniezzo et al.	1994	Quadratic assignment problem
AS-JSP	Colorni et al.	1994	Job shop scheduling problem
Ant-Q	Dorigo, Gambardella	1997	Traveling salesman problem
ACS-3opt and ACS	Dorigo, Gambardella	1997	Traveling salesman problem
ABC	Schoonderwoerd et al.	1996	Telecommunications networks
AS-VRP	Bullnheimer et al.	1997	Vehicle routing problem
MMAS	Socha et al.	2003	University course timetabling

			problem
HAS QAP	Gambardella et al.	1999	Quadratic assignment problem
HAS SOP	Gambardella and Dorigo	2000	Sequential ordering problem
AS-ATP	Costa and Hertz	1997	Graph coloring
ANTCOL	Costa and Hertz	1997	Graph coloring
AntNet & AntNet-FA	Di Caro and Dorigo	1997	Connectionless network routing
Regular ants	Subramanian et al.	1997	Routing in dynamic network
MMAS-QAP	Stutzle and Hoos	2000	Quadratic assignment problem
AS-QAP	Maniezzo and Colorni	1999	Quadratic assignment problem
	Maniezzo	1999	Quadratic assignment problem

ANTS-QAP	Solimanpur et al.	2004	Intercell layout problem in cellular manufacturing
AS-SCS	Michel and Middendorf	1999	Shortest super sequence problem
ASGA	White et al.	1998	Connection management
AntNet-FS	Di Caro and Dorigo	1998	Connection-oriented network routing
ABC-smart ants	Bonabeau et al.	1998	Connection-oriented network routing
CAF	Heusse et al.	1998	Routing networks
ABC-backward	Van der Put	1998	Routing in the faxfactory
	Stutzle	1998	Flow shop problem
	Bland	1999	Space-planning
	Doerner et al.	2003	Full truckload transportation

	Doerner et al.	2000	Pickup and delivery
	Jayaraman et al.	2001	Bioreactors optimization
	Bland	2001	Structural design problem
	Gravel et al.	2002	Scheduling continuous casting
ACO	Roli et al.	2001	Constraint satisfaction
	Gamez and Puerta	2002	Best elimination sequence
	Eggers et al.	2003	Keyboard arrangement problem
	Shelokar et al.	2004	Clustering
	Gandibleux et al.	2004	Set packing problem
	Reimann and Laumanns	2005	Capacitated minimum spanning tree problem
	Lim et al.	2005	Bandwidth minimization
	Maniezzo and	2000	Frequency assignment

	Carbonaro		problem
ANTS			
	Maniezzo et al.	2001	Data warehouse logical design
	Montemanni et al.	2002	Minimum-span frequency
AntNet routing	Bar´an and Sosa	2001	Data networks routing
AC2	Cicirello	2001	Shop floor routing
Anthill	Baboglu et al.	2001	Peer-to-peer (P2P) networks
Parallel ant colonies	Talbi et al.	2001	Quadratic assignment problem
Ant heuristic	McMullen	2001	JIT sequencing problem
Ant-TDVRP	Rizzoli et al.	2002	Vehicle routing problem
ACS-DVRP	Montemanni et al.	2002	Dynamic vehicle routing problem
Intelligent ant	Zhou and Liu	1998	Dynamic routing of telecommunication networks

MACS-VRPTW	Gambardella et al.	1999	Vehicle routing problem
ACSp	Bianchi et al.	2004	Probabilistic traveling salesman problem
API	Monmarch´e et al.	2000	Numeric optimization
BWAS	Cordon et al.	2000	Traveling salesman problem

The combinatorial optimization problem defines the pheromone model of ACO, where a pheromone value is associated with each possible solution component; that is, with each possible assignment of a value to a variable. Formally, the pheromone value τ_{ij} is associated with the solution component c_{ij} , which consists of the assignment $X_i=v_i^j$.

The set of all possible solution components is denoted by C.

Each artificial ant builds a solution through traversing across complete construction graph G(V,E) where V: is a set of vertices and E: a set of edges between these vertices where the set of possible component solution C can be a set of sequence of edges or vertices that have values viewed as pheromone based on some constraints, these solution component constructed incrementally that can be viewed as partial solutions constructing when each ants move on the graph from a vertex to a vertex and deposits a certain amount of pheromone $\Delta\tau$.

To apply real ant behaviors in the ACO, the following is the important components to accomplish the algorithm:

- State transition rule: Each ant moves to the next node to change the trail state by applying a state transition rule. This rule contains two kinds of information, which provide ants to choose the next node with a probabilistic decision. The first is pheromone information, which represents a global view according to the density of pheromone. The second is visibility information, which means a local view according to the cost of edge for state transition. Two parameters are used to control the relative importance between the two kinds of information and are adjusted experimentally.
- Pheromone updating rule: When all ant have constructed their tours, the intensity of pheromone is updated by evaporation, local updating and global updating processes. We know that shorter paths will reserve more amounts of pheromone and the paths will become more attractive in the next iteration. The pheromone updating process can be applied after each ant walks through a node (local updating) or after each ant has constructed the completed tour (global updating). In a real ants society, when fewer ants walk through a path, the pheromone remained on it will become less and less because of evaporation effects. Similarly in the ACS, the evaporation effects make sure that a path, which is not refreshed, becomes less desirable.
- Stopping criterion: The ACO process is iterated until the end condition is met. The stopping criterion of the ACO could be a limited number of iterations, limited executed CPU time or maximum number of iterations within which no better solution is attainable.

The following is a more detailed description of the ACO initialization and the 3 components.

1. Set parameters, initialize pheromone trials by initialize all parameters and pheromone values for each component that will be traversed next; initialized to constant value, that required by ants to start their search.

2. Construct Ant solution, each ant starts building their own solution by traversing the search space S, adding elements incrementally from a finite set of components, C. this construction is like walking through graph edges from vertex to vertex, where the process of selecting a component depends on the following probabilistic Equation (2.1) (Dorigo et al., 1991).

$$P_{ij}^{k} = \begin{cases} \dfrac{[\tau_{ij}(n)]^{\alpha}.[\eta_{ij}]^{\beta}}{\sum_{k \in allowed\ k}[\tau_{ik}(n)]^{\alpha}.[\eta_{ik}]^{\beta}} & if\ j \in allowed\ k; \\ 0, & otherwise. \end{cases} \qquad (2\,)$$

where

τ_{ij} is the amount of pheromone on arc i, j.

α is a parameter to control the influence of τ_{ij} .

η_{ij} is the desirability of arc i,j (a prior knowledge, typically $1/d_{ij}$), d_{ij} represents the distance between i,j.

β is a parameter to control the influence of η_{ij} .

The two parameters α and β may be viewed as positive parameters whose values determine the significant of pheromone information and heuristic information.

This equation favors components with large pheromone values. In most ACO algorithms the probabilities for choosing the next solution component, also called the transition probabilities.

3. Apply Local Search: a local search procedure may be applied for improving the solutions constructed by the ants. The use of such procedure is optional, through experimentally it has been observed that, if available, its use improves the algorithms overall performance.

4. Update pheromones: increases the pheromone values associated good solutions and decreases those associated with bad ones. Usually, this is achieved by decreasing all pheromone values through pheromone evaporation, and by increasing all pheromone levels associated with a chosen set of good solutions. Different ACO algorithms such as, for example, Ant System (AS), Ant Colony System (ACS) and MAX-MIN Ant System (MMAS) mainly differ in the update of the pheromone values they apply as discussed in the following paragraphs.

The main Variants of ACO

1. Ant system

The original algorithm in ACO (Dorigo et al., 1996) where each ant builds a solution update the pheromone values where the pheromoneτ_{ij} , associated with the edge joining vertex i and j,

$$\tau_{ij} \leftarrow (1-\rho).\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}{}^{k} \qquad (3)$$

where

ρ is the evaporation rate.

m is the number of ants.

$\Delta\tau_{ij}$ is the amount of pheromone laid on edge (i,j) by ant k.

$$\Delta\tau_{ij} = \begin{cases} \frac{Q}{l_k}, & if\ ant\ k\ used\ edge\ (i,j) in\ its\ tour; \\ 0, & otherwise. \end{cases} \quad (4)$$

where

Q is a constant

L_k is the length of the tour constructed by ant k.

2. MAX-MIN Ant System (MMAS)

Another improvement algorithm, proposed by Stuzle and Hoos (2000), its characterizing elements are that only the best ant updates the pheromone trails and that the value of the pheromone is bound. The pheromone update is implemented as follows:

$$\tau_{ij} \leftarrow [(1-\rho).\tau_{ij} + \Delta\tau_{ij}^{best}]_{\tau_{min}}^{\tau_{max}} \quad (5)$$

Where τ_{max} and τ_{min} are respectively the upper and lower bounds imposed on the pheromone; the operator $[x]_a^b$ is defined as

$$[x]_a^b = \begin{cases} a, & x > a; \\ b, & x < 0; \\ x & otherwise. \end{cases} \quad (6)$$

Where $\Delta\tau_{ij}^{best}$ is

$$\Delta\tau_{ij}^{best} = \begin{cases} \frac{1}{L_{best}}, & if (i,j) belongs\ to\ the\ best\ tour; \\ 0, & otherwise. \end{cases} \quad (7)$$

3. Ant Colony System

Gambardella and Dorigo proposed in 1995 the Ant-Q algorithm, an extension of AS which integrates some ideas from Q-learning, and in 1996 Ant Colony System (ACS) (Dorigo and Gambardella; 1996) a simplified version of Ant-Q which maintained approximately the same level of performance, measured by algorithm complexity and by computational results. In ACS once all ants have computed their tour (i.e. at the end of each iteration), ants update the pheromone trail using all the solutions produced by the ant colony. Each edge belonging to one of the computed solutions is modified by an amount of pheromone proportional to its solution value. At the end of this phase the pheromone of the entire system evaporates and the process of construction and update is iterated.

In ACS a new state transition rule called *pseudo-random-proportional* is introduced. The *pseudorandom- proportional* rule is a compromise between the *pseudo-random* state choice rule typically used in Q -learning and the *random-proportional* action choice rule typically used in Ant System. With the pseudo-random rule the chosen state is the best with probability q_0 (*exploitation*) while a random state is chosen with probability 1-q_0 (*exploration*).

The ACS pseudo-random-proportional state transition rule provides a direct way to balance between exploration of new states and exploitation of a priori and accumulated knowledge. The probability for an ant to move from vertex i to j depends on a random variable q uniformly distributed over [0,1], and a parameter q_0 ; if $q<q_0$, then , among the feasible components, the component that maximizes the product $\tau_{ij} .\eta_{ij}^{\beta}$ is chosen, otherwise, the same equation as in AS is used. Local pheromone update is performed by

all ants after each construction step. Each ant applies it only to the last edge traversed using the following formula (8):

$$\tau_{ij} = (1 - \varphi).\tau_{ij} + \varphi.\tau_0 \quad (8)$$

where

$\varphi \in (0,1]$ is the pheromone decay coefficient (the evaporation rate).

τ_0 is the initial value of the pheromone.

The main goal of the local update is to diversify the search performed by subsequent ants during one iteration. In fact, decreasing the pheromone concentration on the edges as they are traversed during one iteration encourages subsequent ants to choose other edges and hence to produce different solutions. This makes it less likely that several ants produce identical solutions during one iteration. Additionally, because of the local pheromone update in ACS, the minimum values of the pheromone are limited.

As in AS, the ACS performs at the end of the construction process a pheromone update, called global pheromone update. ACS global pheromone update is performed only by the best ant, only edges that were visited by the best ant are updated, according to the equation (9):

$$\tau_{ij} \leftarrow (1 - \rho).\tau_{ij} + \rho.\tau_{ij}^{best} \quad (9)$$

Where τ_{ij}^{best} $=1/L_{best}$ if the best ant used edge (i,j) in its tour, τ_{ij}^{best} =0 ,otherwise (L_{best} can be set to either the length of the best tour found in the current iteration , L_{ib} or the best solution found since the start of the algorithm, best-so-far, L_{bs}).

2.5 Related Regression Test case Prioritization Techniques

In recent years, several researchers have addressed the test case prioritization problem and suggested techniques for this problem. In this section, we aim to present an overview of these techniques.

1. Rothermel et al. (1999), described several techniques for test case prioritization based on many criteria's. Optimal technique is one of these techniques, that for a given program P and a set of known faults for P, if we determine, for test suite T, which test cases in T expose which faults in P; then this technique can determine an optimal ordering of the test cases in T for maximizing T's rate of fault detection. For this technique, in practice, it is not a practical one since it requires a prior knowledge of existence of faults and which faults. Second technique is total statement coverage prioritization coverage prioritization, it prioritize the test cases based on the total number of statements they cover, by counting the number of statements covered by each test case, and then sorting the test cases in descending order. The cost of this technique is O(*mn*) where *m* is number of test cases, and *n* is the number of total statements in program P. Third technique is additional statement coverage prioritization, it schedules test cases in the order of total coverage achieved, by selecting iteratively a test case that yields the greatest statement coverage, then adjust coverage information on all remaining test cases to indicate their coverage. This process is repeated until all statements covered by at least one test case. The cost of this technique is $O(m^2n)$, so it is more expensive than previous technique. Also techniques depends on branch coverage is supposed.

Total fault-exposing-potential (FEP) prioritization technique is based on the ability of test case to expose a fault, and on the probability on that fault in a statement will

cause a failure. Additional fault-exposing-potential is an extension of total fault-exposing-potential prioritization technique.

2. Wong et al. (1997) suggested prioritization test cases according to the criterion of increasing cost per additional coverage. Prioritization is carried on test cases selected from regression suite that execute any modifications in the old program and refer to this technique as a modification-based test selection technique. Then, depending on the available recourses, a trade- off between what should ideally do in regression testing and what can afford to is applied to determine which tests, should be reexcuted first, and which ones have lower priority. Two techniques are used, test set minimization and prioritization. Although authors didn't specify a mechanism to prioritize the remaining test cases after full coverage has been achieved, their technique reveals faults earlier in the test process.

3. Kim and Porter. (2002) suggested prioritization technique that is History-based test prioritization for regression testing in resource constrained environment. They presented technique that use historical information about test cases. Historical information can be test execution history, or test case fault detection, or/and test case code coverage. These different test histories yield different prioritization, but they increase the effectiveness of long running regression testing process and reduce cost and effort.

4. Aggrawal, et al. (2004) suggest prioritization technique which is Code Coverage Based for Regression Test Case prioritization. Their proposed technique is version specific regression test selection algorithm, interested in executing only those lines of code that have been modified and aims to execute the modified lines of code with minimum number of test cases. Their proposed technique achieved 100% code coverage optimally during version specific regression testing.

5. Rothermel, et al. (2005) have designed and performed a controlled experiment examining whether test case prioritization can be effective on Java programs tested under JUnit, and compared the results to those achieved in earlier studies. Their analysis shows that test case prioritization can significantly improve the rate of fault detection of JUnit test suites, but also reveal differences with respect to previous studies that can be related to the language and testing paradigm.

6. Fazlalizadeh, et al. (2009) proposed equation considers historical effectiveness of the test cases in fault detection. The main contribution of their work is presenting a new equation to compute the priority of test cases in each session of regression testing. The proposed equation considers the environment time and resource constraints and incorporates three factors: historical effectiveness in fault detection, each test case's execution history in regression test and finally the last priority assigned to the test case.

In their equation, for each test case in each test session, historical performance is the proportion of the test case program fails to the number of its executions up to now; this ration is called historical fault detection effectiveness. Another effective factor in priority is a period of time that a test case is not being executed. In other words, it should be ensured that after some sessions, all test cases in test suite will be executed and it will cycle through all test cases over multiple sessions and the related faults will be revealed. Finally, the third factor, shown by empirical studies, is the recent priority of each test case in each test suite during past executions of the regression test. Their empirical studies shows considerable improvements in fault detection rate which was measured using Average Percentage Fault Detection (APFD).

7. Li, et al. (2007) investigated search algorithm for Regression Test Case Prioritization. They presented results of empirical studies for several algorithms; Greedy, Metaheuristic, and Evolutionary search algorithms. The five search algorithms

studied are: two metaheuristic algorithms (Hill Climbing and Genetic Algorithms), together with three Greedy Algorithms (Greedy, Additional Greedy,, and 2-Optimal Greedy). They compared the performance of these algorithms over six different programs. They proved that Genetic Algorithm performs well in test case prioritization.

8. Srivastava and Thiagarajan. (2002) proposed a technique that based on the changes that have been made to the program and focused on the objective function of “impacted block coverage”. They have built a test prioritization system, Echelon, which prioritizes an existing set of tests. The proposed technique is a practical binary code based approach for test prioritization based on program change. Echelon compute changes between programs at a very fine granularity—at the level of basic blocks— using an accurate binary matching algorithm and it uses a fast, simple algorithm that works well in practice and does not attempt an expensive data flow algorithm to determine which new code will be covered by which existing tests.

9. Walcott.k and Kapfhammer .M. (2006) proposed a regression test prioritization technique that uses a Genetic Algorithm to reorder test suites in light of testing time constraints. Their technique prioritize regression test suites so that the new ordering will always run within a given time limit, and will have the highest possible potential for defect detection based on derived coverage information. In their research they compare the results of proposed GA with different parameters, (i) the initial test suite ordering, (ii) the reverse of the initial test suite ordering, (iv) random test suite prioritizations, and (v) fault-aware prioritizations, showing that the GA produced prioritizations are superior to the other suite reordering. In their empirical evaluation reveals that the technique is especially applicable when there is a fixed set of time constraints, or when prioritization occurs infrequently or the time constraint is particularly small.

10. Singh et al. (2010) suggested Test Case Prioritization using Ant Colony Optimization algorithm. Their work presented technique based on Ant Colony Optimization to order test cases with respect to execution time of each test case and its value as the percentage of fault coverage. Their work was not tested with real examples. The results weren’t compared with algorithms in the literature but it was compared with random ordering, optimum ordering and reverse ordering.

Chapter 3: Ant Colony Algorithm to prioritize Regression test case

In this chapter, we will provide an overview of the algorithms employed in this study. Besides, a comprehensive discussion of the work procedure (steps) herein adopted will be outlined. This chapter is divided into two sections.

First section will concentrate on the proposed Technique in the area of regression test case prioritization using ACO especially Ant System. The intended technique aims mainly to choose matrices of test cases and statements, blocks, and decision, then finding the best order of these test cases to reach optimal or near optimal order that covers a high percentage of program code. Also Genetic Algorithm is tested to the same data input and code coverage is computed.

Second section will introduce the main results obtained from both ACO and GA, and an analysis will be discussed.

The main aims behind this chapter are as follows:

1. Programs and test suites selection: Programs from Siemens with its huge testing pool are experiments subjects.
2. The proposed Ant System Settings : to build the proposed technique, a set of previous knowledge should be prepared to be a starting point for the process of the proposed technique as this study depends on ACO especially Ant System.
3. Building Ant System Phases: after finding the needed parameters, Ant System steps which are initialization Pheromone Trails, Construct Ant Solutions, and Update Pheromone are implemented.
4. Studying Genetic Algorithm: as Genetic algorithm is one of evolutionary algorithms that used to solve regression test case prioritization, it is studied in this work to the same input data of AS.

5. Results and Analysis: experiments results are analyzed and compared for the two algorithms.

3.1 Experiments Subjects

In order to improve the generality of the results reported in our thesis, the basic search experiments were instantiated with several values of the three primary parameters that govern the nature and outcomes of the search, namely:

1. The program to which regression testing is applied. Four C programs were subjects, ranging from 374 to 726 lines of code.
2. The coverage criterion to be optimized. The three choices studied were statements, block, and decision coverage for each program.
3. The size of the test suite, that two granularities of test suite size were used including: (i) small suites of size 2-25, and (ii) large suites of size 15-410.

The programs and test suites were from an infrastructure that is designed to support controlled experimentation with software testing regression testing techniques.

Print_tokens2 and Schedule2 are variations of the programs Print_tokens and Schedule, respectively. These four programs were assembled by researchers at Siemens Corporate Research for experiments with control-flow and data-flow test adequacy criteria (Li, et al., 2007).Schedule and Schedule2 are priority schedulers, and Print_tokens and Print_tokens2 are lexical analyzers. For these four programs, the researchers at Siemens created a test pool of black box test cases using the category partition method and TSL tool (Beizer,1990) (Munson and Khoshgoftaar, 1992). They then augmented this test pool with manually created white-box test cases to ensure that

each exercisable statement, edge, and definition-use pair in the base program or its control flow graph was exercised by at least 30 test cases. The researchers also created faulty versions of each program by modifying code in the base version; in most cases they modified between 2 and 5 lines of code. Their goal was to introduce faults that were realistic as possible, based on their experience with real programs. To obtain meaningful results, the researchers retained only faults that were detectable by at least 3 and at most 350 test cases in the associated test pool.

Table (3). Experiments Subjects.

subject	LoC	Blocks	Decisions	Test pool size	Average small test suite size	Average large test suite size
Print_tokens	726	126	123	4,130	20	312
Print_tokens2	570	103	154	4,115	12	350
Schedule	412	46	56	2,650	19	286
Schedule2	374	53	74	2,710	10	310

In our thesis we divide the experiments into two categories: small test suits, and large test suites. In order to produce small test suites, a test case is selected at random from the test pool and is added to the test suite only if it adds to the cumulative statements or block or decision coverage depending on the test evaluation criteria. This is repeated until the test suite achieves full coverage. In order to produce large test suites, test cases are randomly selected from the test pool and added to the test suite until full coverage is achieved.

Both small and large test suites are binary matrices encoding, that if test case covers statement (or block or decision), the cell in the matrix is one otherwise it is zero. Therefore, we have 50 small test suites for block coverage, another 50 small test suites for statements coverage, and 50 small test suites for decision coverage. For large test suites we generate 25 test suites selected from testing pool for each criteria statements coverage, decision coverage, and block coverage.

Table (4). Example of test matrix.

Test Case	Statement							
	1	2	3	4	5	6	7	8
A	X	X	X			X	X	X
B	X	X	X				X	X
C	X	X	X	X				
D					X	X	X	X

Test Case	Statement							
	1	2	3	4	5	6	7	8
A	1	1	1	0	0	1	1	1
B	1	1	1	0	0	0	1	1
C	1	1	1	1	0	0	0	0
D	0	0	0	0	1	1	1	1

3.2 The Proposed Ant System

The importance of the original Ant System (AS) resides in being a prototype of a number of ant algorithms which collectively implement the ACO paradigram. AS already follows the same behavior of ACO with some specified elements as will discussed in the following.

Algorithm 1. Ant Colony optimization metaheuristic

```
Set parameters, initialize pheromone trials
while termination condition not met do
   ConstructAntSolutions
   ApplyLocalSearch      {optional}
   UpdatePheromone
End while
```

For our combinatorial problem; test cases prioritization, given n-test cases with ω_k , the artificial ants are published and distributed to the test cases randomly. Each ant decides independently which test case to select next, until the tour is completed. At the beginning of iteration, all test cases can be selected except the one the ant is located in can. Each ant has to visit each test case exactly once; the visited (selected) test cases have to be stored.

The following is more detailed description of initialization and 3 phases of AS.

3.2.1 Initial Settings

It is common with the application of ACO to find some “tuning” of the parameters is required to determine the values that yield the best results, so some initial experiments were performed to find the values of parameters that will give results better than other parameters values. Different values for ants’ number, iteration number, α, β, and ρ were tested, and the following values gave good results.

a. Ants’ number: ants’ number is set to 100 for small test suites size, and for large test suites ants’ number is 260.

b. Iteration number: iteration for small test suites it is set to 50 and 200 for large test suites.

c. α : the parameter to control the influence of τ_{ij} is set to 1 for all experiments.

d. β: the parameter to control the influence of η_{ij} is set to 5 for all experiments.

e. ρ: the evaporation rate is 0.1 for all experiments.

f. Initial pheromone: before ants start traveling to visit test cases and building traces, initial pheromone trials are set for all test cases equally according to the following equation:

$$\tau_0(t_i) = 1/(n) \tag{10}$$

τ_0 : initial pheromone for test cast t_i .

n :total number of test cases in the test suite

3.2.2 Ants Lifecycle

Ants go in a continuous cycle of forage and storage, until all test cases are collected where new program iteration starts.

1. Initial ants placing

Before beginning construction solutions, each ant of the colony select random test case from the search space, *n*. These selected test cases are the staring nodes for the solutions.

2. Construct Ant Solution

At each iteration of the algorithm, each ant moves from a test case *i* to test case *j*, corresponding to a more complete intermediate solution. Thus, each ant *k* computes a set *Ak(i)* of feasible expansions to its current state in each iteration, and moves to one of

these in probability. For ant k, the probability P_{ij}^{k} of moving from test case *i* to test case *j* depends on the combination of two values, the attractiveness η_{ij} of the move, as computed by some heuristic indicating the a priori desirability of that move and the trail level τ_{ij} of the move, indicating how proficient it has been in the past to make that particular move.

According to the following probability formula, ants select test cases to be next node traversed.

$$P_{ij}^{k} = \begin{cases} \dfrac{[\tau_{ij}(n)]^{\alpha} . [\eta_{ij}]^{\beta}}{\sum_{k \in allowed\ k} [\tau_{ik}(n)]^{\alpha} . [\eta_{ik}]^{\beta}} & if\ j \in allowed\ k; \\ 0, & otherwise. \end{cases} \quad (11)$$

τ_{ij} is the amount of pheromone on test case *i*.

α is a parameter to control the influence of τ_{ij}

η_{ij} is the heuristic value of test case *i* (a priori knowledge).

β is a parameter to control the influence of η_{ij} .

Heuristic function is a prior knowledge given to ants as matrix, calculated as:

$$\eta \ = h_1 .* h_2 \quad (12)$$

where

1. h_1 used to give the test case t_i a heuristic value that is inversely proportional to the number of test cases that cover the same statement .

$$h_1\,(t_i)=\sum_{i=1}^{statements}\ \sum_{j=1}^{n}\left(\frac{1}{\sum_{i=1}^{statements}\ a(i,j)}\right) \qquad (13)$$

2. h_2 used to give test case that covers statements which are not covered be previously selected test cases.

   coverage(ti) function: we define a function ‘coverage’ which gives each test case in the un-prioritized test suite the set of statements it covers.

$$h_2(t_i)=Count\ (Coverage(ti)-\ \bigcap_{k=1}^{i-1} Coverage\ (t_k))+0.0001 \qquad (14)$$

c. Ants tour cost calculating

After ants complete its tour, each solution built is evaluated in order to know good solutions. For our proposed AS algorithm to order regression test case, solutions cost (fitness function) is computed as the solution average coverage:

$$f(k)=APBC\ or\ (APDC,APSC) \qquad (15)$$

f(k) is the total cost of ant *k* tour’s

Considering the APBC metric as an example and a test suite *T* containing *n* test cases that cover a set *B* of *m* blocks. Let TB_i be the first test case in the order *T* 'of *T* that covers block *i*. The APBC for order *T'* is given by the equation:

$$APBC=1-\frac{TB_1+TB_2+\cdots+TB_m}{nm}+\frac{1}{2n} \qquad (16)$$

APBC measures the weighted average of the percentage of block coverage over the life of the suite. ABPC values range from 0 to 100.

d. Update pheromones

Increases the pheromone values associated with good solutions and decreases those associated with bad ones. Usually, this is achieved by decreasing all the pheromone values through pheromone evaporation, and by increasing the pheromone levels associated with a chosen set of good solutions. In AS, pheromone values are updated as:

$$\tau_{ti} = (1-\rho).\tau_{ti} + \sum_{k=1}^{m} \Delta\tau_{ti}{}^{k} \qquad (17)$$

where ρ is the evaporation rate.

m is the number of ants.

$\Delta\tau_i{}^k$ is the amount of pheromone laid on test case t_i by ant *k*.

For our problem, $\Delta\tau_i{}^k$ is calculated as follows:

$$\Delta\tau_{ti}{}^{k} = \left(f(k).* \frac{h_0(t_i)}{\sum_{j=1}^{statements} j}\right)^{1/indx} \qquad (18)$$

where

$$h_0(t_i) = \sum_{j=1}^{statement} a(i,j) \qquad (19)$$

Where $h_0(t_i)$ is summation of statements covered by test case n_i, and $a(i,j)$ is element of the binary matrix of *i* rows (test cases) and *j* columns (statements or block or decision).

indx is index of test case in the tour of ant *k*.

e. Ants Cycle Termination

The algorithm terminates when a maximum number of iteration has been produced. For heuristic algorithms, if the algorithm has terminated due to a maximum number of generations or iterations, a satisfactory solution may or may not have been reached.

The following is AS algorithm proposed for regression test case prioritization:

```
1. Initialization
     Initialize  τ_ti  and η_ti ,  ∀ i ∈ t;
2. Construction
   For iter=1 to Number of Iterations do
     Place each ant k on initial test case
        For k=1 to m (# ants)do
          Repeat until ant k has completed a tour
              Select the test case i to be visited next
              With probability P_ij  given by equation (11)
      Calculate the fitness function f(k)of the tour generated by
      ant k  given by equation (15)
 Update the trail levels τ_ti  on all test cases according (17)
 End For (build one route)
```

Algorithm 3.2. AS procedure steps.

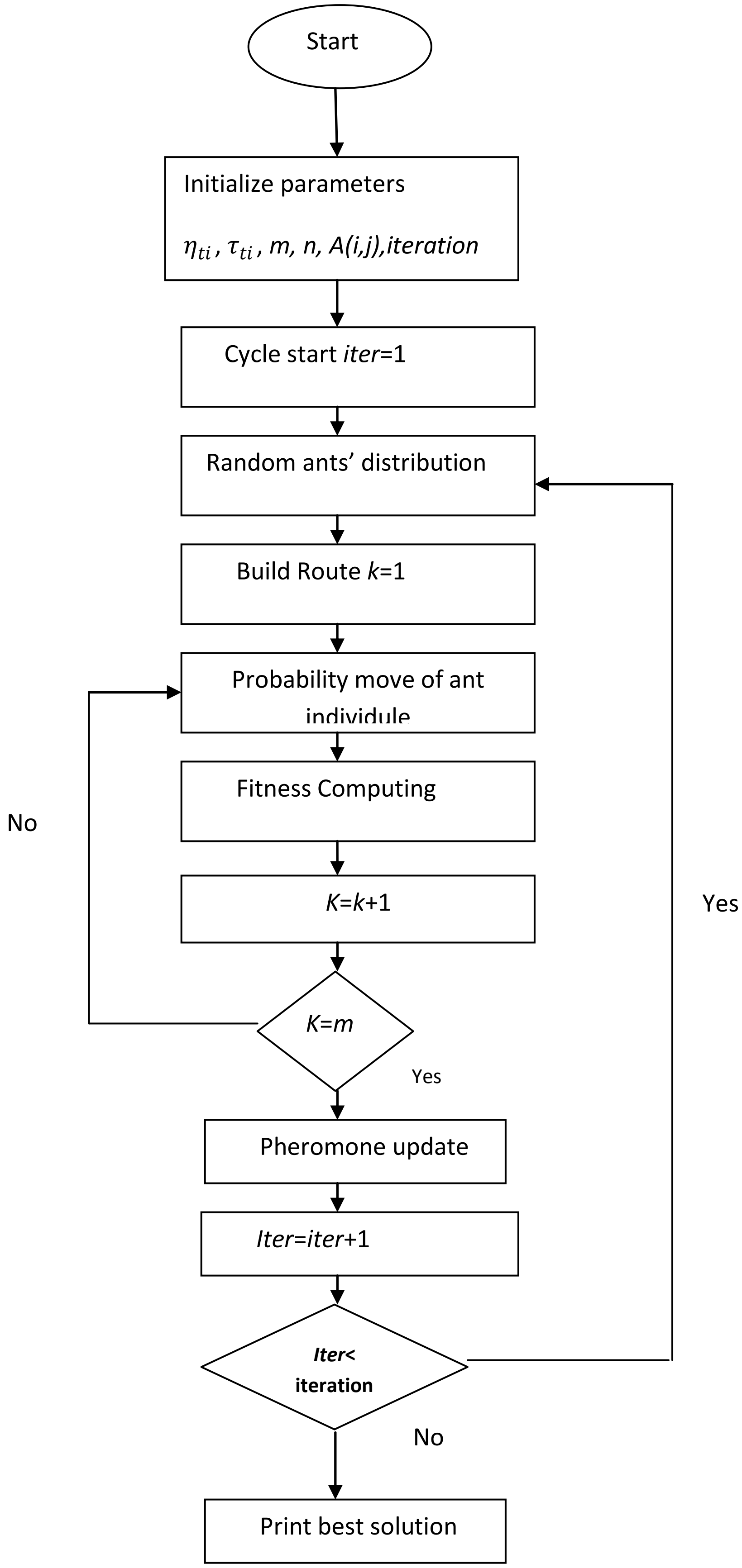


Figure (7). Flow Chart of AS

3.3 Genetic Algorithm

Genetic algorithms (GAs) represent a class of adaptive search techniques based on the evolutionary ideas of natural selection and genetic. The basic concept of GAs is designed to simulate processes in natural system necessary for evolution, specifically those that follow the principles first laid down by Charles Darwin of survival of the fittest. Algorithms has been widely studied, experimented and applied in many fields in engineering worlds. Not only does GAs provide an alternative method to solving problem, it consistently outperforms other traditional methods in most of the problems link. Many of the real world problems involved finding optimal parameters, which might prove difficult for traditional methods but ideal for Gas (Goldberg, 1989).

In a genetic algorithm, a population of strings (called chromosomes), which encode candidate solutions (called individuals) to an optimization problem, evolves toward better solutions. Solutions are represented in many forms, binary as strings of 0s and 1s, real-valued, or character-based. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

Algorithm 2. Genetic Algorithm procedure

```
begin
iter ← 0
Initialize P(iter)
Evaluate P(iter)
while (not termination condition) do
  begin
   iter ← iter + 1
   Select P(iter) from P(iter-1) according to evaluation
   Crossover P(iter) according to crossover rate
   Mutate P(iter) according to mutation rate
   Evaluate P(iter)
  end
end
```

The detailed settings for the parameters and the operators of the GA in the experiments reported in this thesis are described as follows:

1. Encoding: Binary encoding is not suitable for the sequencing problem, so an ordering chromosome was used. For a test suite with *n* test cases, the chromosome is encoded as *n*-sized array, and the value of a chromosome element indicates the position of a test case in the order.
2. Initializing: Initially many individual solutions are randomly generated to form an initial population. Initial population has a set of prioritized test cases solutions. The population size depends on the size of the test suites. For small test suites, population size is set to 60 and number of iteration is 50, and for large test suites population size is 260 and iteration number is 200.

3. Selecting: is the stage of a genetic algorithm in which individual genomes are chosen from a population for later breeding (recombination or crossover). In our problem, selection is based on fitness function computed for each individual (test case order). Fitness function is based on code coverage. Parents with highest fitness function are selected $\{T_{p1}, T_{p2}\}$. Fitness function computed for individuals is as following:

$$APBC = 1 - \frac{TB_1 + TB_2 + \cdots + TB_m}{nm} + \frac{1}{2n} \qquad (16)$$

4. Crossover (Recombination): Crossover is used to vary population from one test case prioritization set to the next through recombination. It is unlikely that the new population after recombination will be identical to previous parents set P*(iter-1).* A single crossover point on both parents' organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. First, random number is generated $r \in n$ to be crossover point for each parent. The subsequences before and after the crossover point are then exchanged to produce two new offspring, as present in Figure (8).

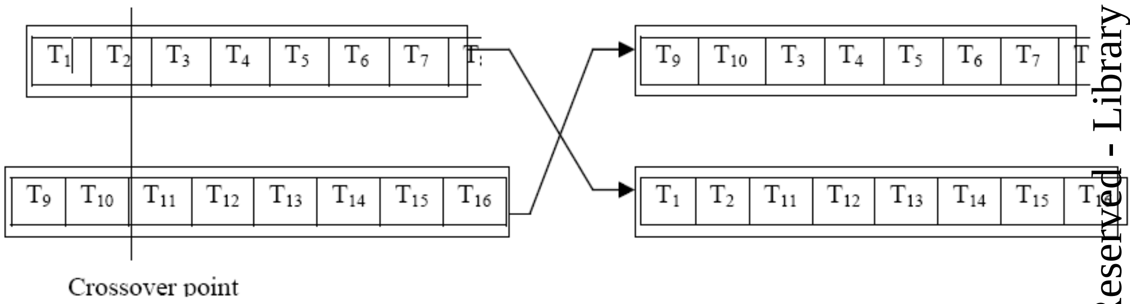


Figure (8). Crossover with Random Crossover Point.

5. Mutation: a mutation operator involves a probability that an arbitrary bit in a genetic sequence will be changed from its original state. Mutations are swaps, inversions and scrambles. For our work, we generate random number $r2 \in n$

indicate mutation point where flip or swap or slide operation will carried on for that point. Figure (9).

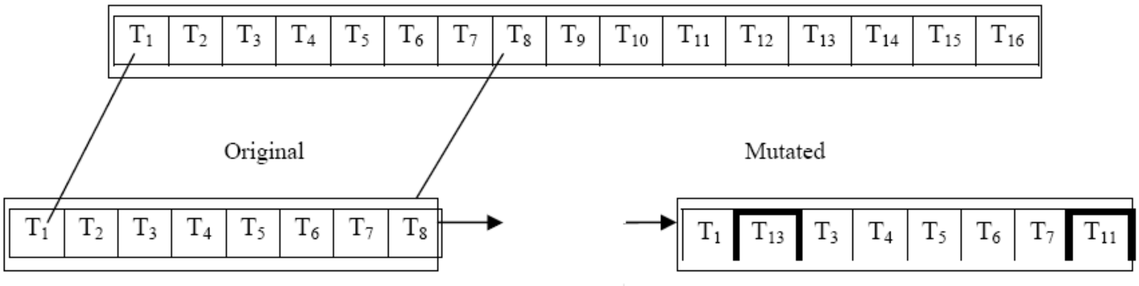


Figure (9). Mutation by swapping two test cases.

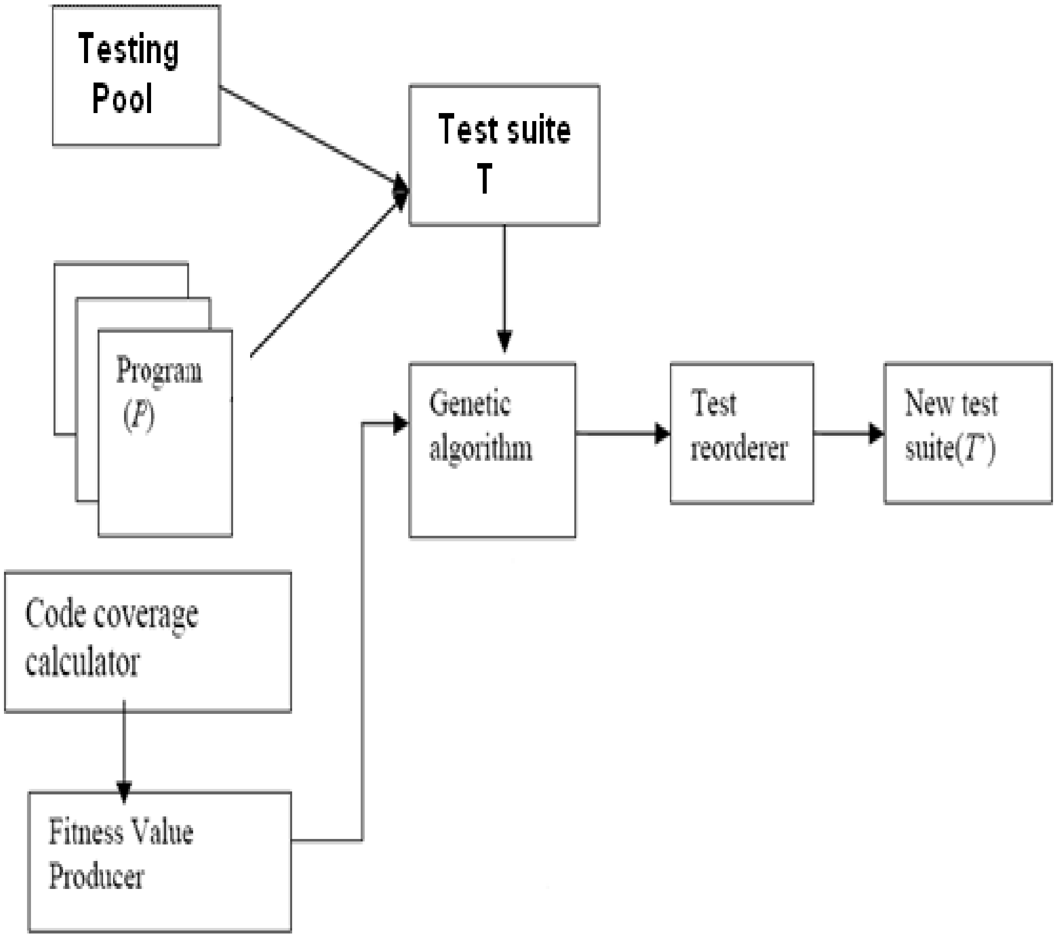


Figure (10). Over view of GA prioritization.

3.4 Results and comparisons

In this section, the proposed algorithms AS experimental results are introduced in addition with the results of Genetic Algorithm. These results are processed and analyzed using SPSS. SPSS is professional statistical software used to generate box plots and perform T-test to investigate the statistical significance of any differences observed in the experiments.

The fitness metric of AS and GA will be shown and discussed.

3.4.1 Predefined Assumption

Paired Samples T-test analysis is an advanced statistical technique, provides a statistical test of whether or not the means of two groups are all equal. This T-test is used when the sample are dependent, that is, when there is one sample that has been test twice.

Paired Samples T-test calculates the difference between all pairs of the sample. Significance *P-value* reports the significance level, the smaller the *P-value*, the stronger the evidence against the null hypothesis.

In order to analyze experimental data, null hypothesis (H_O: $\mu_1=\mu_2=\mu_3$) is set that the means of the fitness values (APBC, APDC, and/or APSC) for the two algorithms are equal. In order to decide whether to accept or reject the null hypothesis, the level of significance of the results was set at 0.1, thus the confidence level is 90%.

3.4.2 Experiments with Small Test Suites

1. Schedule program

The results of the three fitness metrics studied for Schedule program tests are:

a. Average Percentage Block Coverage (APBC): the results of 50 experiments proved that the proposed AS Algorithm is significantly better than Genetic Algorithm at 90% of confidence interval, since significant difference between algorithms is less than 0.1. As show in Figure (11).

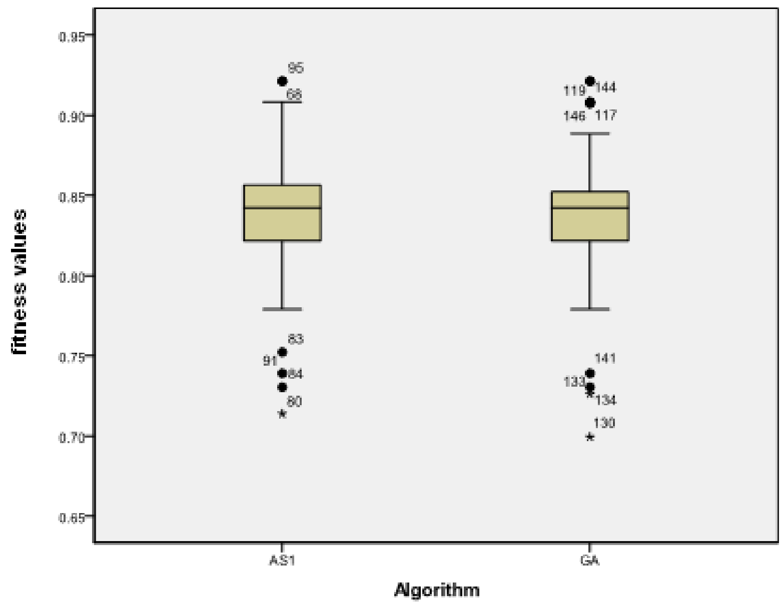


Figure (11). Box Plots of APBC for Schedule program with small test suites.

b. Average Percentage Decision Coverage (APDC): the results of 50 experiments proved that there is no significant difference between proposed AS Algorithm and Genetic Algorithm at 90% of confidence interval. Figure (12).

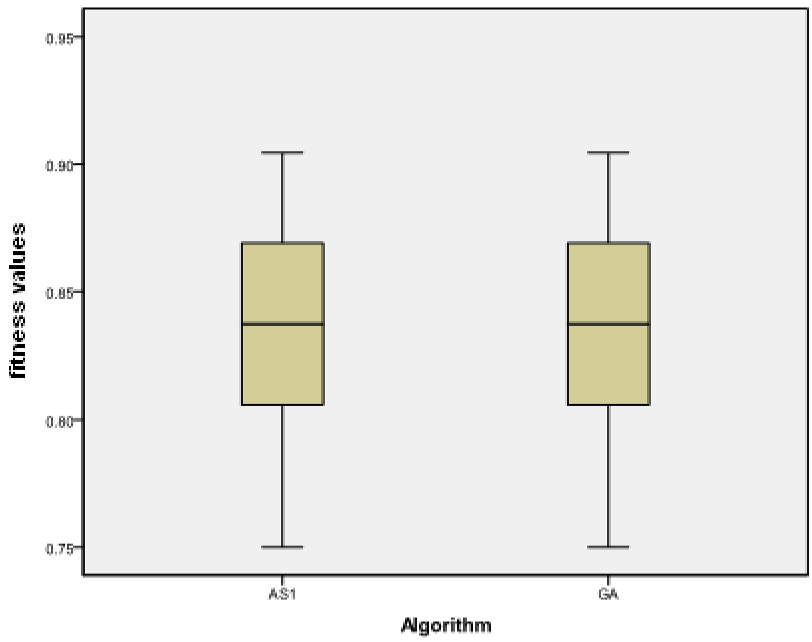


Figure (12). Box Plots of APDC for Schedule program with small test suites.

c. Average Percentage Statement Coverage (APSC): for the 50 tests, the results show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. Figure (13).

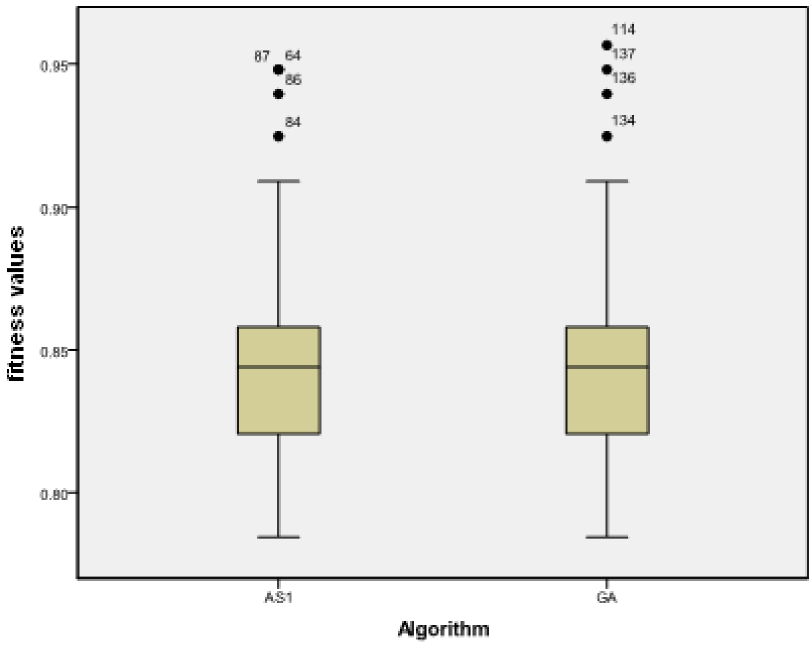


Figure (13). Box Plots APSC for Schedule program with small test suites.

2. Schedule2 program

a. Average Percentage Block Coverage: for 50 tests, the results show that Ant System Algorithm is significantly better than Genetic Algorithm at 90% of confidence level, since the significant difference is less than (0.1). Figure (14).

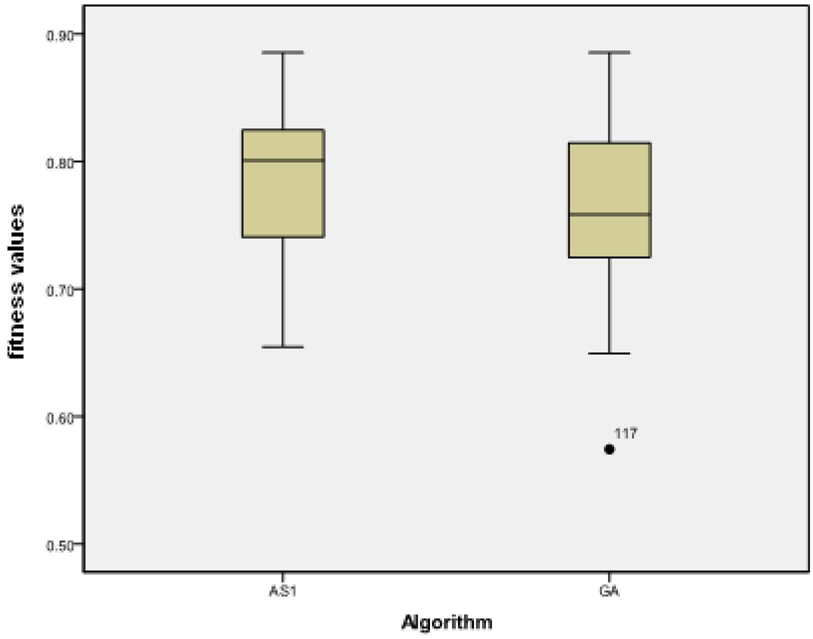


Figure (14). Box Plots of APBC for Schedule2 program with small test suites.

b. Average Percentage Decision Coverage: the results for 50 tests proved that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. Figure (15).

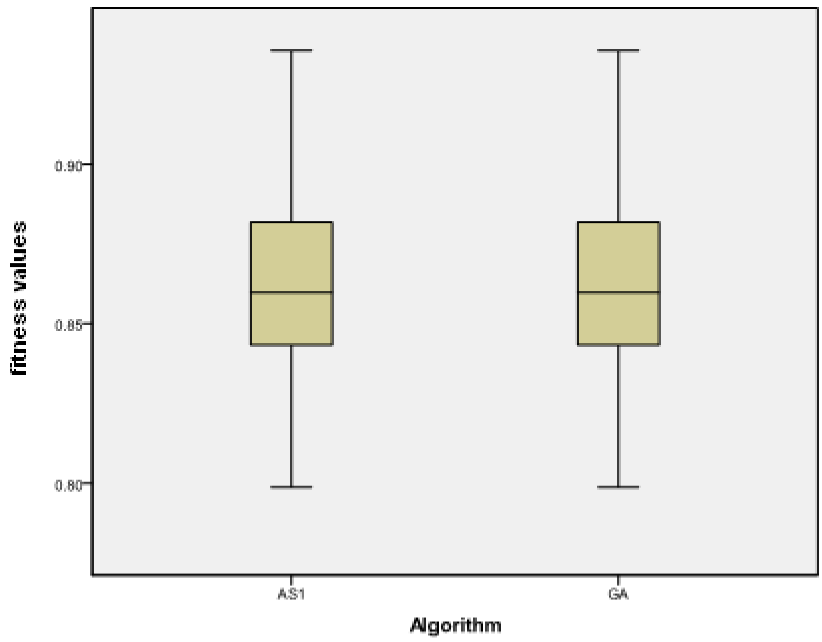


Figure (15). Box Plots of APDC of Schedule2 with small test suites.

c. Average Percentage Statement Coverage: the 50 results show that Ant System Algorithm is significantly better that Genetic Algorithm at 90% of confidence level, since the significant difference value is less than (0.1).

Figure (16).

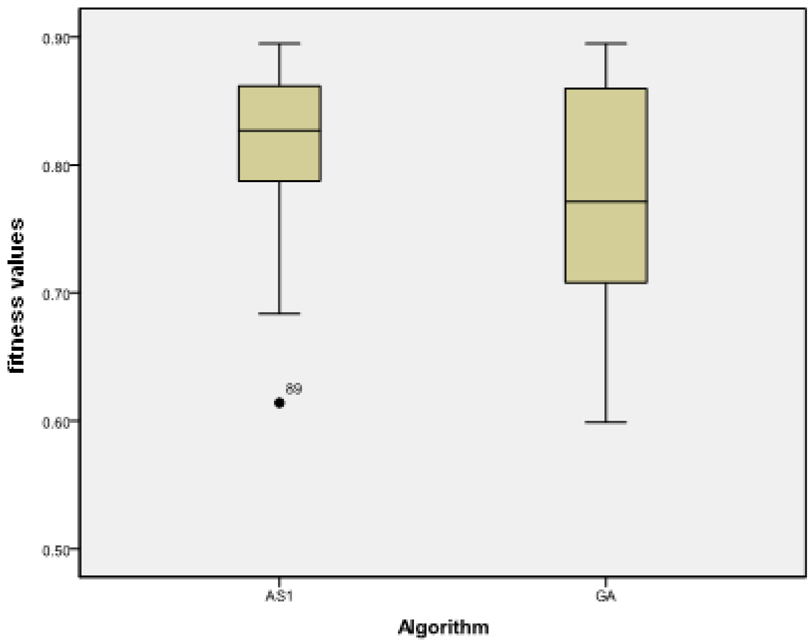


Figure (16). Box Plots of APSC of Schedule2 program with small test suites.

3. Printtokens program

a. Average Percentage Block Coverage: for Printtokens program, the 50 tests prove that Genetic Algorithm is significantly better than Ant System algorithm at 90% of confidence level, since the significant difference value is less than (0.1). Figure (17).

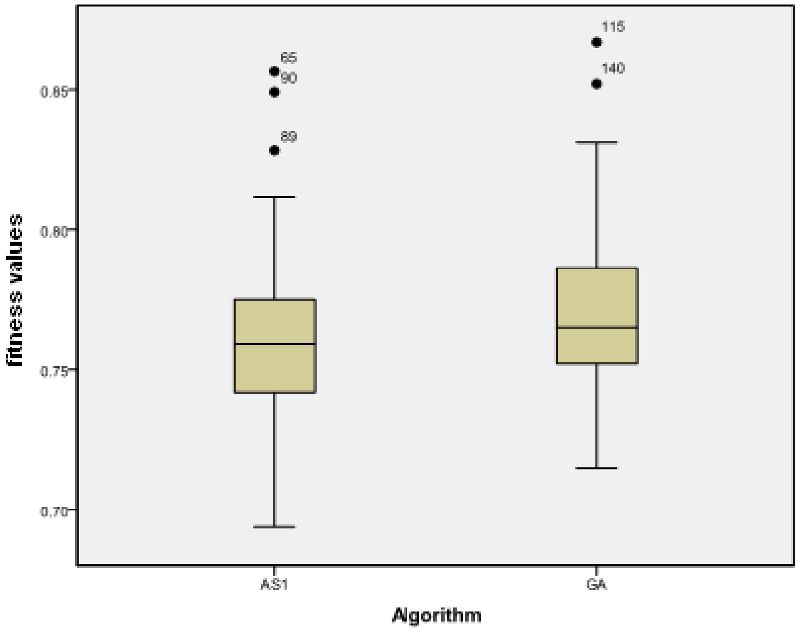


Figure (17). Box Plots of APBC for Printtokens program with small test suites.

b. Average Percentage Decision Coverage: for Printtokens program, the 50 tests prove that Genetic Algorithm is significantly better than Ant System algorithm at 90% of confidence level, since the significant difference value is less than (0.1). Figure (18).

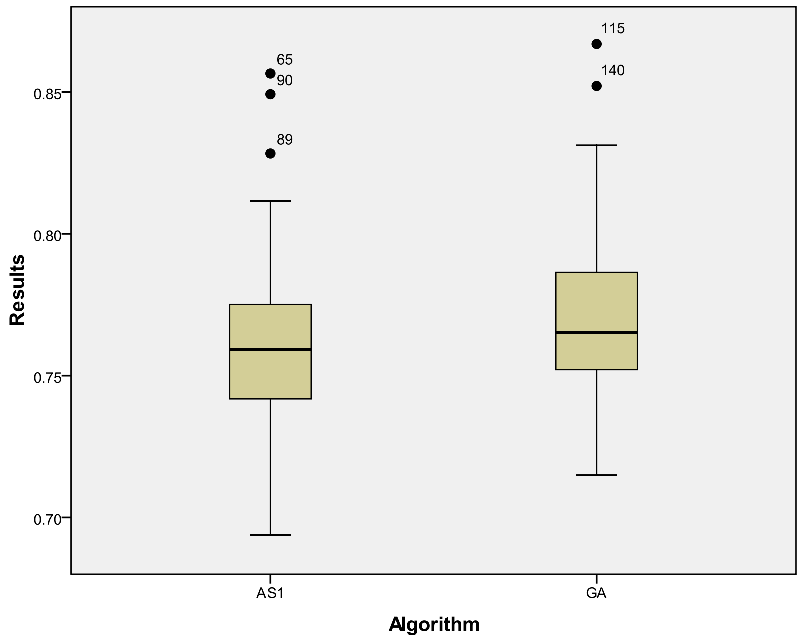


Figure (18). Box Plots of APDC of Printtokens program with small test suites.

c. Average Percentage Statement Coverage: the 50 tests prove that Genetic Algorithm is significantly better than Ant System algorithm at 90% of confidence level, since the significant difference value is less than (0.1). Figure (19).

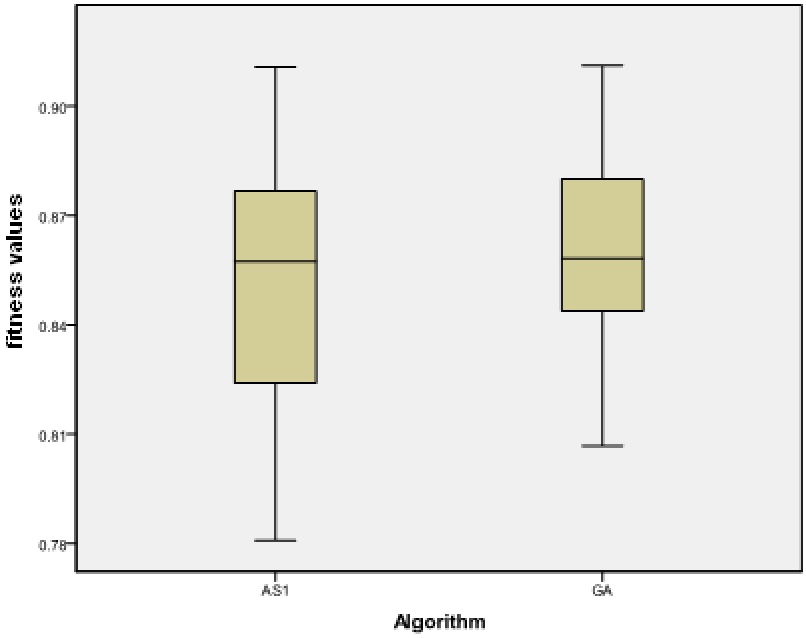


Figure (19). Box Plots of APSC of Printtokens program with small test suites.

4. Printtokens2 program

a. Average Percentage Block Coverage: the results of 50 tests show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. As shown in Figure (20).

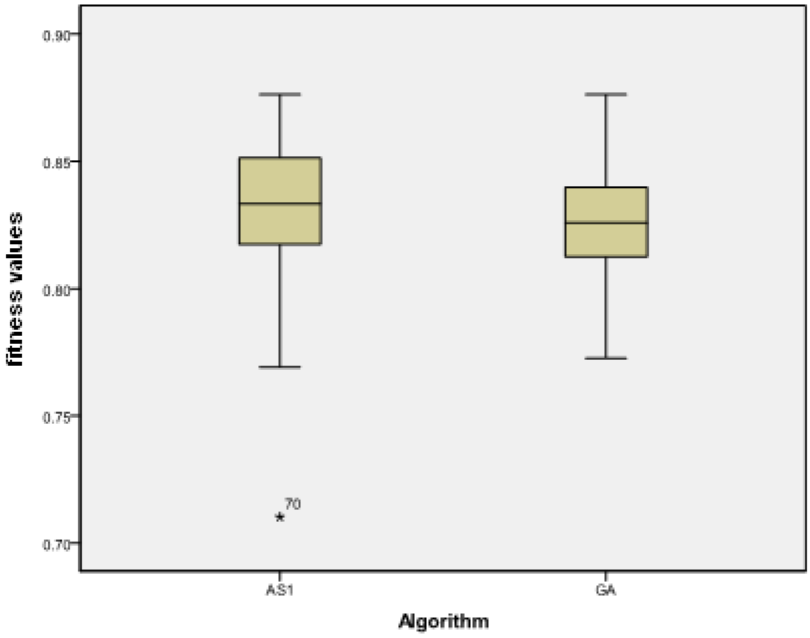


Figure (20). Box Plots of APBC Printtokens2 program with small test suites.

b. Average Percentage Decision Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (21).

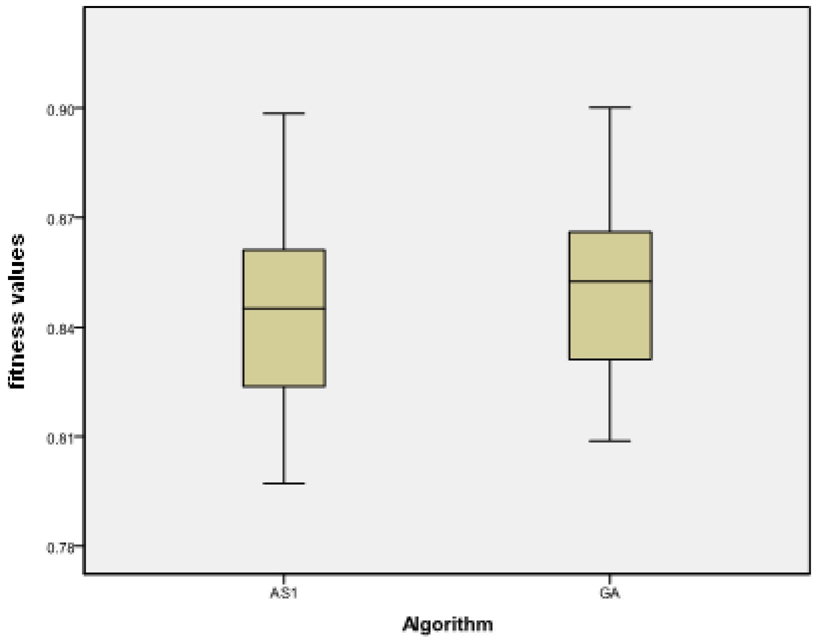


Figure (21). Box Plots of ABDC of Printtokens2 with small test suites.

c. Average Percentage Statement Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (22).

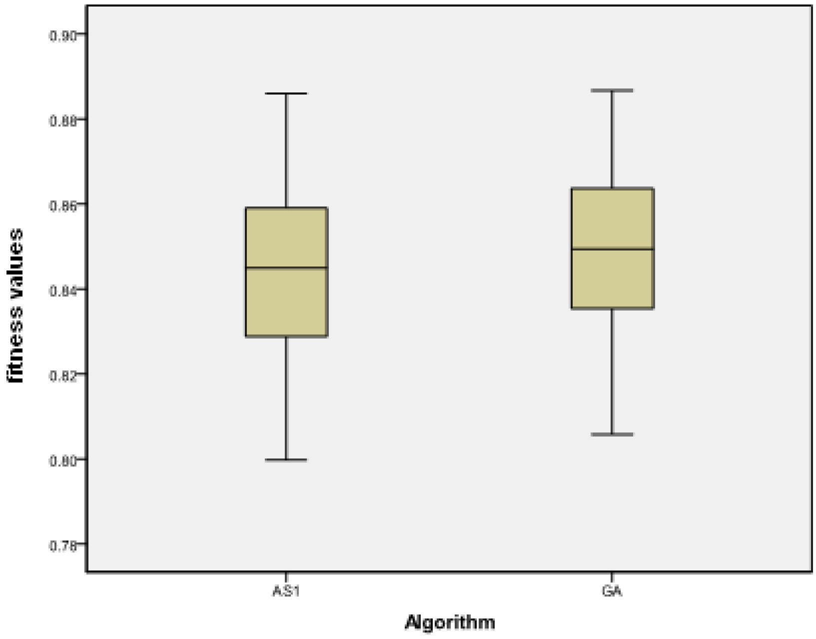


Figure (22). Box Plots of APSC of Printtokens2 program with small test suites.

3.4.3 Experiments with Large Test suites

1. Schedule program

   a. Average Percentage Block Coverage: the results of 25 tests show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. As shown in Figure (23).

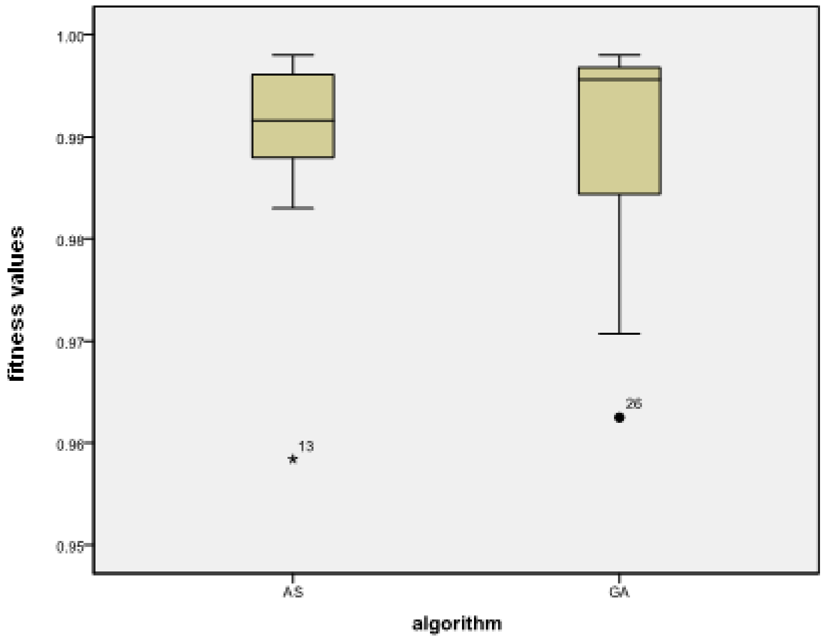


Figure (23). Box Plots of APBC of Schedule program with large test suites.

   b. Average Percentage Decision Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (24).

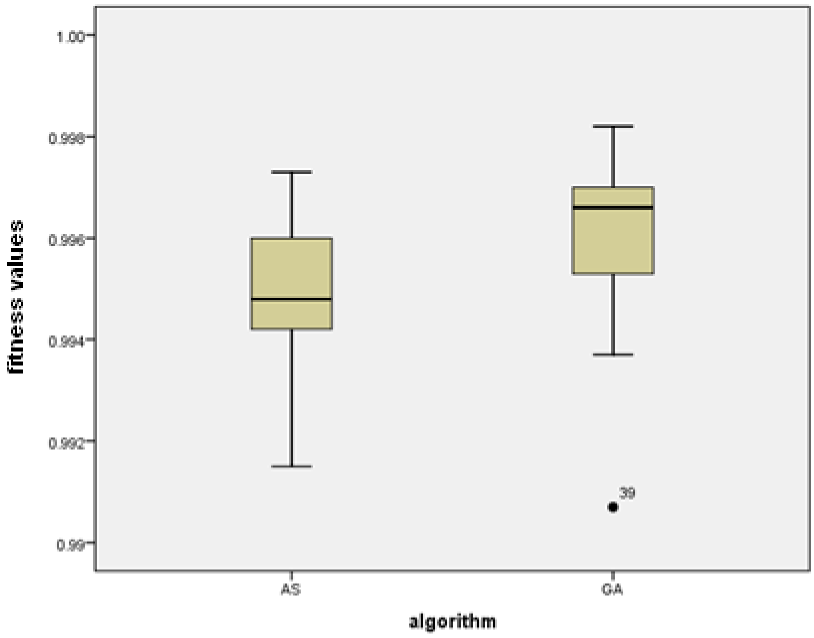


Figure (24). Box Plots of APDC of Schedule program with large test suites.

c. Average Percentage Statement Coverage: the results of 25 tests show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. As shown in Figure (25).

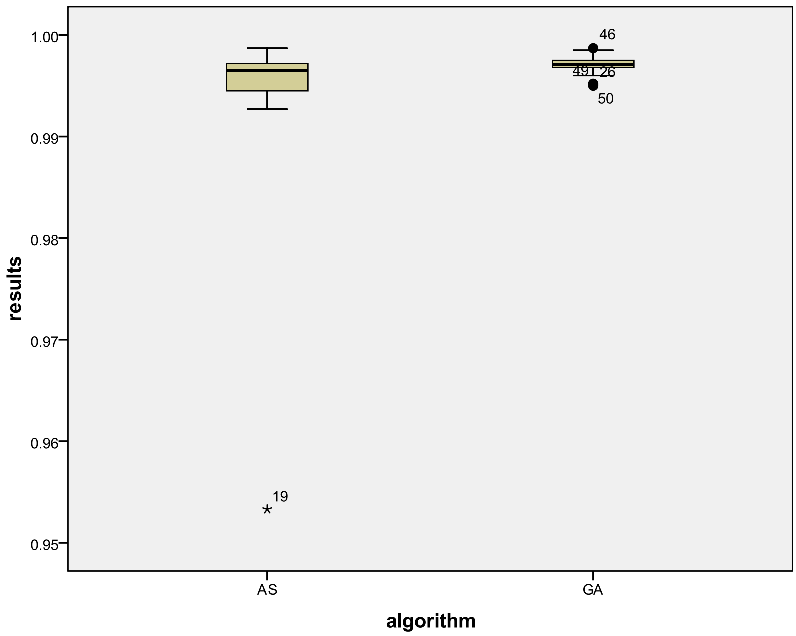


Figure (25). Box Plots of APSC of Schedule program with large test suites.

2. Schedule2 program

   a. Average Percentage Block Coverage: the results show that Ant System Algorithm is significantly better than Genetic Algorithm at 90% of confidence level. As shown in Figure (26).

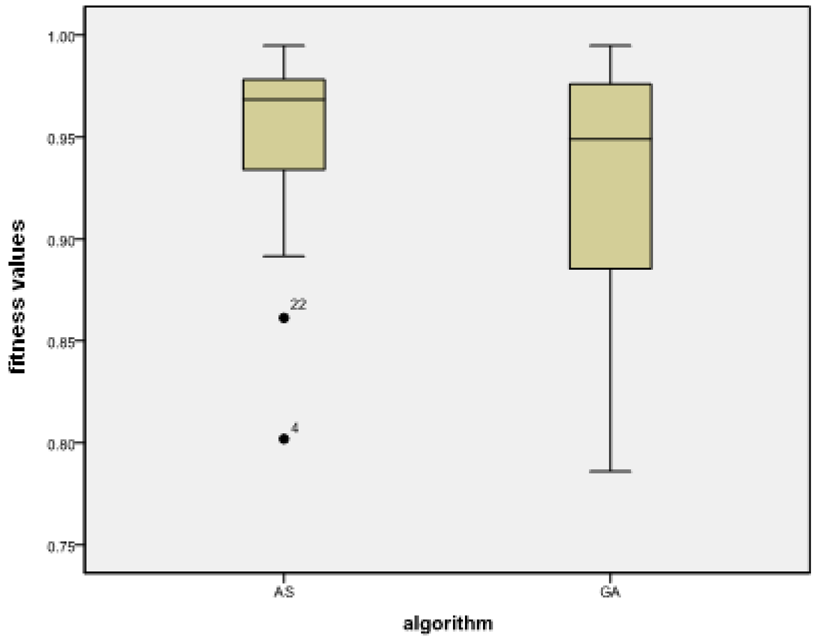


Figure (26). Box Plots of APBC of Schedule2 program with large test suites.

   b. Average Percentage Decision Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (27).

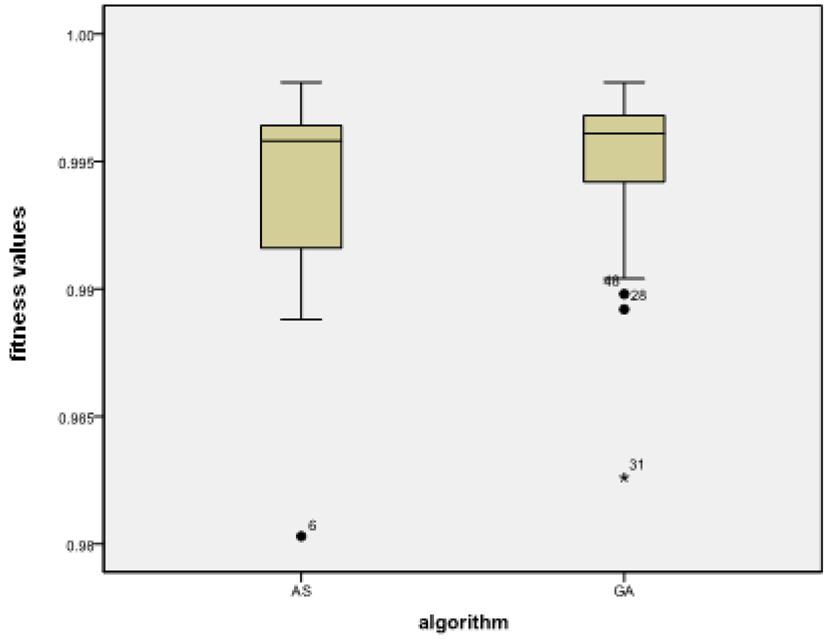


Figure (27). Box Plots of APDC of Schedule2 program with large test suites.

c. Average Percentage Statement Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (28).

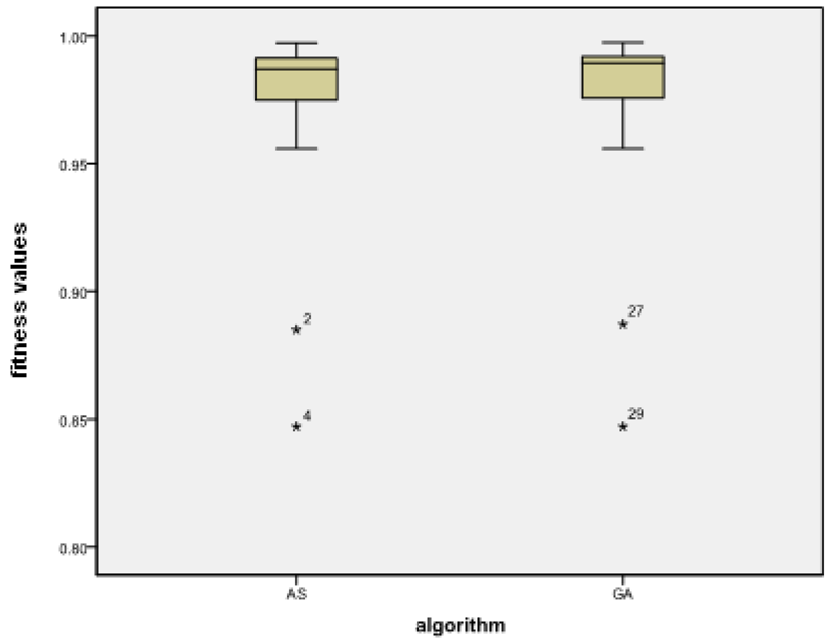


Figure (28). Box Plots of APSC of Schedule2 program with large test suites.

3. Printtokens program

a. Average Percentage Block Coverage: the results of 25 tests show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. As shown in Figure (29).

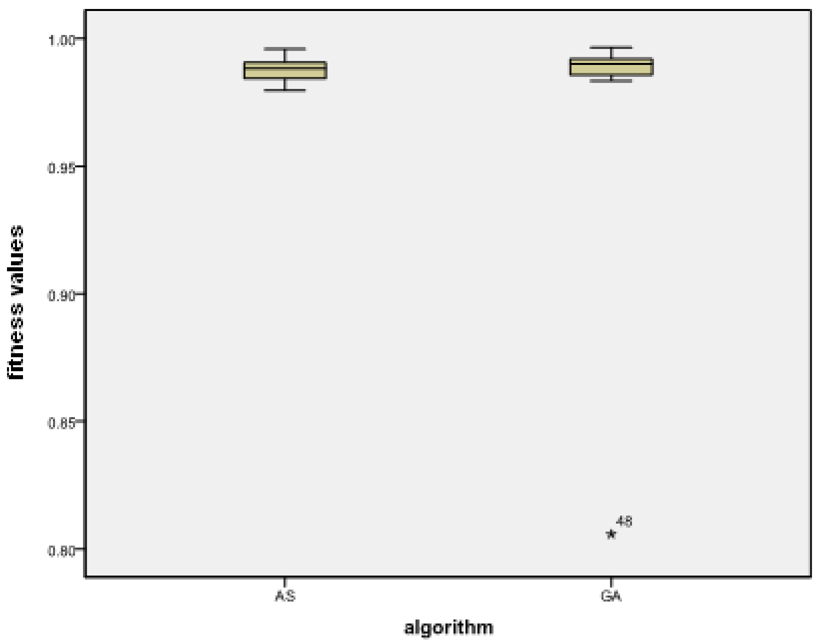


Figure (29). Box Plots of APBC of Printtokens program with large test suites.

b. Average Percentage Decision Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (30).

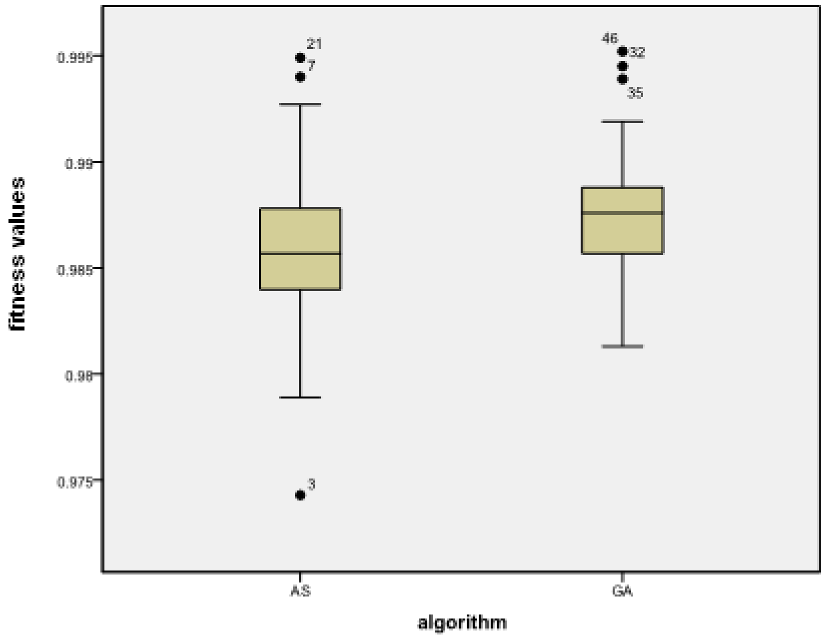


Figure (30). Box Plots of APDC of Printtokens program with large test suites.

c. Average Percentage Statement Coverage: the results show that Genetic Algorithm is significantly better than Ant System Algorithm at 90% of confidence level. As shown in Figure (31).

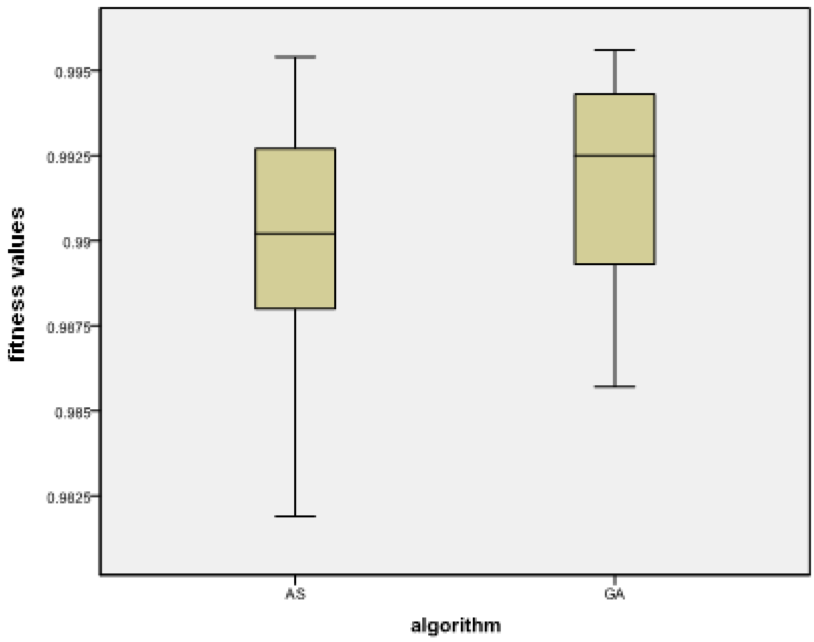


Figure (31). Box Plots of APSC of Printtokens with large test suites.

4. Printtokens2 program

    a. Average Percentage Block Coverage: the results show that Ant System Algorithm is significantly better than Genetic Algorithm at 90% of confidence level. As is shown in Figure (32).

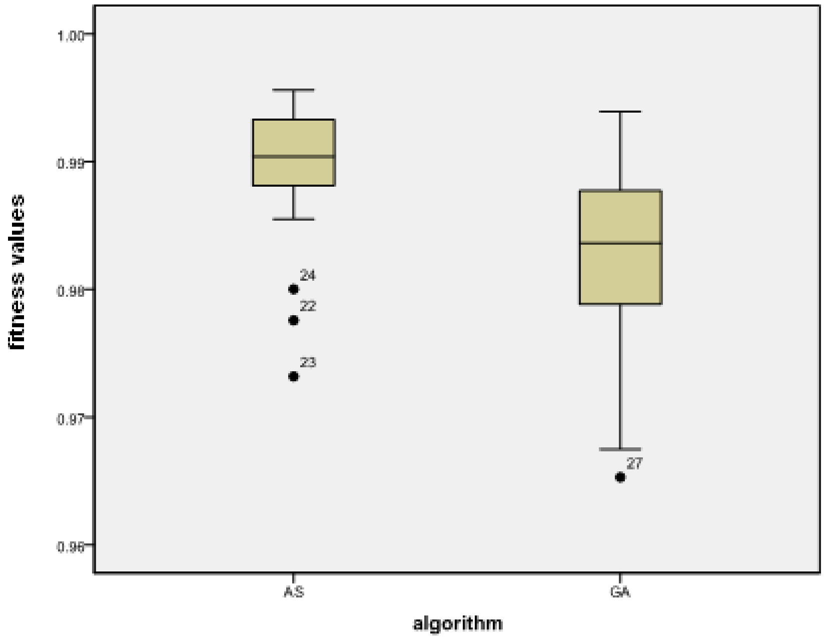


Figure (32). Box Plots of APBC of Printtokens2 with large test suites.

    b. Average Percentage Decision Coverage: the results of 25 tests show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. As shown in Figure (33).

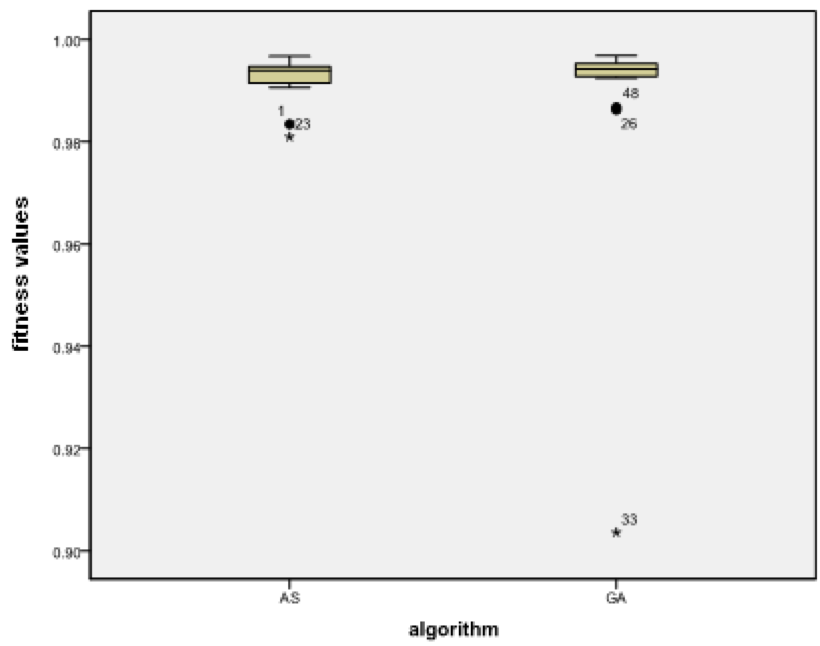


Figure (33). Box Plots of APDC of Printtokens2 program with large test suites.

c. Average Percentage Statement Coverage: the results of 25 tests show that there is no significant difference between Ant System Algorithm and Genetic Algorithm at 90% of confidence level. As shown in Figure (34).

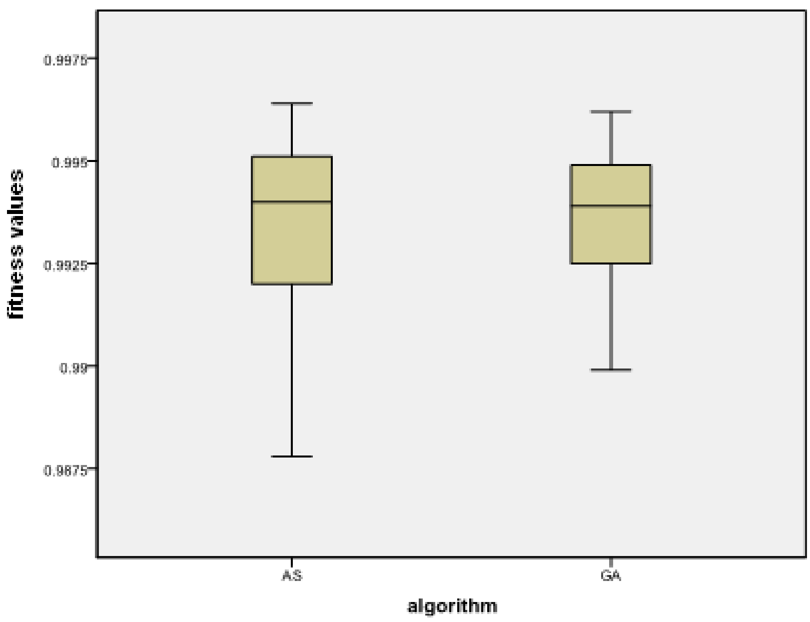


Figure (34). Box Plots of APSC of Printtokens2 program with large test suites.

Table (5). Summary of results for all programs with small test suites.

program	Results with 90% confidence level	
Schedule	Block	AS is better than GA
	Decision	No significant difference
	Statement	No significant difference
Schedule2	Block	AS is better than GA
	Decision	No significant difference
	Statement	AS is better than GA
Printtokens	Block	GA is better than AS
	Decision	GA is better than AS
	Statement	GA is better than AS
Printtokens2	Block	No significant difference
	Decision	GA is better than AS
	Statement	GA is better than AS

Table (6). Summary of results for all programs with large test suites.

program	Results with 90% confidence level	
Schedule	Block	No significant difference
	Decision	GA is better than AS
	Statement	No significant difference
Schedule2	Block	AS is better than GA
	Decision	GA is better than AS
	Statement	GA is better than AS
Printtokens	Block	No significant difference
	Decision	GA is better than AS
	Statement	GA is better than AS
Printtokens2	Block	AS is better than GA
	Decision	No significant difference
	Statement	No significant difference

Chapter 4: Conclusion and Future Work

4.1 Conclusion

The ACO metaheuristic was inspired by the foraging behavior of real ants and is characterized as being a distributed, stochastic search method based on the indirect communication of a colony of artificial ants, mediated by artificial pheromone trails.

In the ACO approach, one key element is the indirect form of memory of previous performance. This role is played by the pheromone matrix. The first basic algorithm was Ant System (AS) and has been introduced using the TSP as an initial application.

The ACO metaheurisitc can be applied to any combinatorial optimization problem for which a solution construction can be conceived. An interesting case is the Regression Test Case Prioritization problem, not only of because its importance in the software testing but also of because the lack of heuristic algorithms used to solve such problem.

Regression Test Case prioritization aims to select all test cases with a priority for the most effective ones. This method does not eliminate some test cases that occur in RTS but it takes a long time to run all test cases although they are ordered.

In this work, we tried in a very simple way to define a new algorithms based on AS for sequencing problem in test case prioritization for regression testing. It presented the results of an empirical study that investigated the effectiveness of AS with a comparison of results of GA. However, the experiments indicate that, in terms of effectiveness, the results show that the two techniques complement each other, in the sense that some ACO techniques achieved significantly better results on some bench mark programs, and GA-based technique achieved significantly better results on some other programs. So, both techniques can be used to order test cases, and fitness function can be used to determine the better order.

4.2 Future Work

Several improvement and direction are possible as future work:

1. This thesis studied the use of AS to prioritize Regression test case based on code coverage, which is different from criteria based on fault detection. Fault detection can possibly yield different results and can be studied as a new criterion with AS algorithm.
2. Combining two methods to prioritize Regression test case such s using Genetic Algorithm (GA) to select test cases from testing pool, and ACO to prioritize these cases.
3. Generating test cases using GA and using ACO to prioritize them.
4. AS is a variant of ACO. Other variants can be studied for our problem such as Ant Colony System (ACS) or MAX-MIN Ant System.
5. Trying other different initial values for ACO techniques.

References

Agrawal, H .Horgan, J.R .Krauser, E and London, S. (1993), 'Incremental regression testing', **Proceedings of the Conference on Software Maintenance**, pp348-357.

Beizer, B. (1990), '**Software Testing Techniques',** Second Edition, Van Nostrand Reinhold Company Limited, ISBN 0-442-20672-0.

Beni, G. Wang, J. (1989), 'Swarm Intelligence in Cellular Robotics System', **NATO Advanced workshop on Robots and Biological Systems,** pp.26-30.

Bonabeau, E. Theraulaz, G. Deneubourg, J.L. Aron, S. and Camazine, S. (1997), 'Self-organization in social insects', **Trends in Ecology and Evolution**, Vol. 12, issue 5,pp. 188-193.

Bonabeau, E. Dorigo, M and Theraulaz, G. (1999), **' Swarm Intelligence :from natural to artificial systems',** First Edition, Oxford University Press, USA, ISBN 0195131592. pp.9-11

Camazine, S. Deneubourg, J. Franks, N. Sneyd, J. Theaulaz, G and Bonabeau, E. (2003), **'Self-Organaization in Biological Systems',** Princeton University Press, ISBN 0-691-11624-5. Chapter 1, page 8.

Deneubourg and Gross (1990), 'Collective patterns and decision making', **Ethology, Ecology and Evolution** 1,295-311.

Dorigo, M. Maniezzo, V and Colorni, A. (1991),' Positive feedback as a search strategy'. **Technical Report 91-016**, Dipartimento di Elettronica, Politecnico di Milano, Milan, Italy.

Dorigo, M, Maniezzo, V and Colorni, A. (1996), “Ant System: Optimization by a colony of cooperating agents”. **IEEE Transactions on Systems**, Man, and Cybernetics – Part B, 26(1):29–41.

Dorigo, M and Gambardella, L.M. (1997), ‘Ant colony system: a cooperative learning approach to the traveling salesman problem’, **IEEE Transaction on Evolutionary Computation,** pp. 53-66.

Do, H. Rothermel, G and Kinner, A. (2003), ‘Prioritizing Junit Test cases: An Empirical Assessment and cost – Benefit Analysis’**, IEEE Transactions on software Engineering.**

Elbaum, S.Malishevsky, A and Rothermel, G . (2001), ‘Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization’, **Proc. 23rd Int’l Conf. Software Engineering**, pp. 329-338.

Elbaum, S. Malishevsky, A.G and Rothermel, G. (2002), ‘Test Case Prioritization: A Family of Empirical Studies’, **IEEE Transaction On Software Engineering**, Vol. 28, No. 2, pp. 159-182.

Fazlalizadeh, Y. Khalilian, A. Abdollahi , M and Parsa, S. (2009), ‘Prioritizing Test Cases for Resource Constraint Environments Using Historical Test Case Performance Data’, **Proc. 2nd IEEE International Conference on Computer Science and Information Technology**, pp.190-195.

Gambardella, L.M and Dorigo, M. (1995), ‘Ant-Q: a reinforcement learning approach to the travelling salesman problem’, **Proceedings of the Twelfth International Conference on Machine Learning.**

Gambardella, L.M and Dorigo, M. (1996), 'Solving Symmetric and Asymmetric TSPs by Ant Colonies', **Proceedings of the IEEE Conference on Evolutionary Computation**, pp. 622-627.

Goldberg, D. E. (1989), 'Genetic Algorithms in Search, Optimization, and Machine Learning', Addison-Wesley, ISBN 0201157675

Grasse', P.P. (1959), 'La reconstruction du nid et les coordinations interindividuelles chez Bellicosutermes netelensis et Cubitermes sp. La theorie de la stigmergie: Essai s'interpretation du comportement des termites constructerurs', **Insects Sociaux** 6,41-83

Haput, R.L. and Haput, S.E. (2004), '**Practical Genetic Algorithms with CD-ROM',** Wiley-Interscience.

Hetzel, W and Hetzel. B. (1991), '**The complete Guide to Software Testing'**, John Willey and Sons, ISBN-0471565679.

Kim, J. M and Porter, A. (2002), 'A History-Based Test Prioritization Technique for regression Testing in Resource Constrained Environment,' **Proc.24th Int'l Conf. Software Engineering**. pp.119-129.

Korel, B . Koutsogiannakis, G and Tahat, L. H. (2007), 'Model-based test prioritization heuristic methods and their evaluation', **Proc. 3rd Int'l Workshop on Advances in Model-Based Testing**, London, UK.

Lewis, M and George, A. (1992), 'The Behavioral Self-Organization of Nan robots using Local Rules', **Proceeding of the IEEE International Conference on Intelligent Robots and Systems.**

Li, Z . Harmanand, M and Hierons, R. (2007), 'Search Algorithms for Regression Test Case Prioritization'**, Proceeding in IEEE Transactions on Software Engineering**, Vol. 33, pp. 225-237.

lu luo. 'Software Testing Techniques', **Technology Maturation and Research International**, Institute for Software Research International, Carnegie Mellon University, PA 152322, USA.

Megen. R and Meyerhoff. D. (1995), 'Costs and Benefits of early detection: experiences from developing client server and host applications', **Software Quality Journal**, Vol. 4, No.4.

Munson, J.C and Khoshgoftaar. T.M. (1992), 'The Detection of Fault Prone Programs', **IEEE Transaction for Software Engineering.**, Vol.18, No.5, pp. 423-433.

Onoma,A.K. Tsai, W.Poonawala, M.H, and H.Suganuma. (1998), 'Regression Testing in an Industrial Environment', **Communications of the ACM**, Vol.41, No.5, pp.81-86.

Park, H. Ryu, H and Baik, J. (2008),' Historical Value-Based Approach for Cost-Cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing', **2th Int'l Conf. Secure System Integration and Reliability Improvement**, pp.39-46.

Rothermel, G and M. J. Harrold, A Safe. (1997), 'Efficient Regression Test Selection Technique', **ACM Transaction on Software Engineering and Methodology**, Vol.6, No.2, pp.173-210.

Rothermel,G. Untch, R. Chu. C and Harrold. M.J. (2001), 'Prioritizing Test Cases for Regression Testing', **IEEE Transaction On Software Engineering**, Vol. 27, No. 10, pp. 929-948.

Rothermel, G. Elbaum, S. Malishevsky, A. Kallakuri, P and Qiu, X. (2004), 'Test Suite Composition and Cost-Effective Regression Testing', **ACM Transactions on Software Engineering and Methodology,** Vol.13, No.3, p.p277-331.

Singh, Y. Kaur, A and Suri, B. (2010), 'Test Case Prioritization using Ant Colony Optimization', **ACM SIGSOFT Software Engineering Notes**, VOL.35, NO.4, pp.1-7

Srivastava. A and Thiagarajan, J. (2002), 'Effectively prioritizing tests in development environment', **Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis** , pp. 97–106

Walcott .K and Kapfhammer.G. (2006), 'Time-Aware Test Suite Prioritization', **Proc. Of the 2006 International Symposium on Software testing and analysis**, pp.1-11.

Wong, W. E . Horgan, J. R. London, S and Ageawal, H. (1997),'A study of effective regression testing in practice'**, Proc. 8th Int'l Symptoms On Software Reliability Engineering**. p.p.230-238.

Wikipedia, (2011).Retrieved in February,from **http://en.wikipedia.org/wiki/ Swarm_intelligence #cite_ref-9**

Appendix A: T-Test Analysis

T-Test for small test suites

1. Schedule Program

a. Schedule program, block based coverage

Table A.1: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8392	50	.04528	.00640
	GA	.8369	50	.04763	.00674

Table A.2: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.990	.000

Table A.3: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
					90% Confidence Interval of the Difference				
		Mean	Std. Deviation	Std. Error Mean	Lower	Upper			
Pair 1	AS - GA	.00222	.00695	.00098	.00058	.00387	2.264	49	.028

b. Schedule program, decision based coverage

Table A.4:Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8367[a]	50	.03561	.00504
	GA	.8367[a]	50	.03561	.00504

The correlation and t cannot be computed because the standard error of the difference is 0.

c. Schedule program, statement based coverage

Table A.5: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8432	50	.04059	.00574
	GA	.8433	50	.04106	.00581

Table A.6: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	1.000	.000

Table A.7: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	-.00017	.00122	.00017	-.00046	.00012	-1.000	49	.322

2. Schedule2 Program

a. Schedule2 program, block based coverage

Table A.8: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.7850	50	.05825	.00824
	GA	.7670	50	.06532	.00924

Table A.9: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS& GA	50	.755	.000

Table A.10: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
				Lower	Upper			

Table A.10: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
				Lower	Upper			
Pair 1 AS - GA	.01795	.04379	.00619	.00757	.02834	2.899	49	.006

b. Schedule2 program, decision based coverage

Table A.11: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8585[a]	50	.02688	.00380
	GA	.8585[a]	50	.02688	.00380

The correlation and t cannot be computed because the standard error of the difference is 0.

c. Schedule2 program, statement based coverage

Table A.12: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8181	50	.05866	.00830
	GA	.7736	50	.08595	.01216

Table A.13: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.216	.131

Table A.14: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
				Lower	Upper			
Pair 1 AS - GA	.04457	.09299	.01315	.02252	.06661	3.389	49	.001

3. Printtokens Program

a. Printtokens program, block based coverage

Table A.15: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.7611	50	.03279	.00464
	GA	.7715	50	.03067	.00434

Table A.16: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.929	.000

Table A.17: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
					90% Confidence Interval of the Difference				
		Mean	Std. Deviation	Std. Error Mean	Lower	Upper			
Pair 1	AS - GA	-.01045	.01210	.00171	-.01332	-.00758	-6.106	49	.000

b. Printtokens program, decision based coverage

Table A.18: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8159	50	.03512	.00497
	GA	.8237	50	.03046	.00431

Table A.19: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.989	.000

Table A.20: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1 AS - GA	-.00772	.00675	.00095	-.00932	-.00612	-8.091	49	.000

c. Printtokens program, statement based coverage

Table A.21: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8536	50	.03301	.00467
	GA	.8611	50	.02724	.00385

Table A.22: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS& GA	50	.948	.000

Table A.23: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1 AS - GA	-.00748	.01129	.00160	-.01015	-.00480	-4.683	49	.000

4. Printtokens2 Program

a. Printtokens2 program, block based coverage

Table A.24: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8303	50	.02874	.00407
	GA	.8265	50	.02294	.00324

Table A.25: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.755	.000

Table A.26: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	.00379	.01889	.00267	-.00069	.00827	1.419	49	.162

b. Printtokens2 program, decision based coverage

Table A.27: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8424	50	.02261	.00320
	GA	.8484	50	.02050	.00290

Table A.28: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.981	.000

Table A.29: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	-.00605	.00473	.00067	-.00717	-.00493	-9.054	49	.000

c. Printtokens2 program, statement based coverage

Table A.30: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.8442	50	.02193	.00310
	GA	.8493	50	.01999	.00283

Table A.31: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	50	.983	.000

Table A.32: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1	AS - GA	-.00512	.00434	.00061	-.00615	-.00409	-8.349	49	.000

T-Test for large test suites

1. Schedule Program

a. Schedule program, block based coverage

Table A.33: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9908	25	.00809	.00162
	GA	.9901	25	.01010	.00202

Table A.34: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.199	.341

Table A.35: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1	AS - GA	.00075	.01162	.00232	-.00323	.00472	.322	24	.750

b. Schedule program, decision based coverage

Table A.36: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9950	25	.00136	.00027
	GA	.9960	25	.00162	.00032

Table A.37: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.673	.000

Table A.38: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	-.00102	.00123	.00025	-.00144	-.00060	-4.137	24	.000

c. Schedule program, statement based coverage

Table A.39: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9944	25	.00874	.00175
	GA	.9971	25	.00100	.00020

Table A.40: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.301	.143

Table A.41: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	-.00266	.00849	.00170	-.00557	.00024	-1.569	24	.130

2. Schedule2 Program

a. Schedule2 program, block based coverage

Table A.42: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9507	25	.04596	.00919
	GA	.9276	25	.05845	.01169

Table A.43: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.718	.000

Table A.44: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
					90% Confidence Interval of the Difference				
		Mean	Std. Deviation	Std. Error Mean	Lower	Upper			
Pair 1	AS - GA	.02310	.04089	.00818	.00911	.03709	2.825	24	.009

b. Schedule2 program, decision based coverage

Table A.45: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9941	25	.00405	.00081
	GA	.9947	25	.00357	.00071

Table A.46: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.956	.000

Table A.47: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
					90% Confidence Interval of the Difference				
		Mean	Std. Deviation	Std. Error Mean	Lower	Upper			
Pair 1	AS - GA	-.00066	.00122	.00024	-.00107	-.00024	-2.680	24	.013

c. Schedule2 program, statement based coverage

Table A.48: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9751	25	.03510	.00702
	GA	.9761	25	.03509	.00702

Table A.49: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.999	.000

Table A.50: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1	AS - GA	-.00098	.00154	.00031	-.00151	-.00046	-3.188	24	.004

3. Printtokens Program

a. Printtokens program, block based coverage

Table A.51: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9875	25	.00427	.00085
	GA	.9822	25	.03694	.00739

Table A.52: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.288	.163

Table A.53: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1	AS - GA	.00534	.03594	.00719	-.00695	.01764	.743	24	.464

b. Printtokens program, decision based coverage

Table A.54: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9858	25	.00469	.00094
	GA	.9879	25	.00360	.00072

Table A.55: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.940	.000

Table A.56: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference Lower	90% Confidence Interval of the Difference Upper			
Pair 1	AS - GA	-.00212	.00180	.00036	-.00273	-.00151	-5.903	24	.000

c. Printtokens program, statement based coverage

Table A.57: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9899	25	.00372	.00074
	GA	.9915	25	.00331	.00066

Table A.58: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.927	.000

Table A.59: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
				Lower	Upper			
Pair 1 AS - GA	-.00160	.00140	.00028	-.00208	-.00112	-5.723	24	.000

4. **Printtokens2 Program**

a. Printtoken2 program, block based coverage

Table A.60: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9897	25	.00565	.00113
	GA	.9825	25	.00720	.00144

Table A.61: Paired Samples Correlations

	N	Correlation	Sig.
Pair 1 AS & GA	25	.440	.028

Table A.62: Paired Samples Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
				Lower	Upper			
Pair 1 AS - GA	.00716	.00692	.00138	.00479	.00953	5.172	24	.000

b. Printtokens2 program, decision based coverage

Table A.63: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9927	25	.00362	.00072
	GA	.9901	25	.01821	.00364

Table A.64: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.164	.432

Table A.65: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	.00261	.01797	.00359	-.00354	.00876	.727	24	.474

c. Printtokens2 program, statement based coverage

Table A.66: Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	AS	.9933	25	.00222	.00044
	GA	.9937	25	.00158	.00032

Table A.67: Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	AS & GA	25	.822	.000

Table A.68: Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	90% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	AS - GA	-.00031	.00129	.00026	-.00075	.00013	-1.209	24	.238

Appendix B: Experiments Results

In this appendix, the results of AS and GA for the four programs are listed for reference.

Results of small matrices

1. Schedule Program

Table B.1. Schedule small block results

AS	GA
0.8424	0.8424
0.8424	0.8424
0.8424	0.8424
0.8424	0.8424
0.8424	0.8424
0.8424	0.8424
0.8522	0.8522
0.8565	0.8565
0.8522	0.8522
0.8217	0.8217
0.8188	0.8188
0.8188	0.8188
0.8188	0.8188
0.7826	0.7826
0.8424	0.8424
0.8424	0.8424
0.9082	0.9082
0.9212	0.9212
0.9076	0.9076
0.8886	0.8886
0.8478	0.8292
0.8587	0.8587
0.8478	0.8478
0.8261	0.8261
0.8152	0.7862
0.779	0.779
0.8261	0.8333
0.8261	0.8261
0.8478	0.8478
0.7138	0.6993
0.8732	0.8732
0.8522	0.8522
0.7522	0.7261
0.7304	0.7304
0.8739	0.8739
0.8424	0.8424

0.8424	0.8424
0.8424	0.8424
0.8424	0.8424
0.8424	0.8424
0.7391	0.7391
0.8565	0.8449
0.8886	0.8886
0.9082	0.9082
0.9212	0.9212
0.9076	0.9076
0.8478	0.8292
0.8261	0.8261
0.8152	0.8152

Table B.2 Schedule small decision result

AS	GA
0.869	0.872
0.8661	0.8661
0.8661	0.8661
0.8661	0.8661
0.8071	0.8071
0.8357	0.8357
0.8357	0.8357
0.8357	0.8357
0.8595	0.8595
0.9045	0.9045
0.8786	0.8786
0.8357	0.8357
0.8143	0.8143
0.8214	0.8214
0.8786	0.8786
0.8036	0.8036
0.8316	0.8316
0.7976	0.7976
0.8138	0.8138
0.7653	0.7653
0.7857	0.7857
0.7934	0.7934
0.8326	0.8326
0.846	0.846
0.8728	0.8728
0.8464	0.8464
0.8036	0.8036
0.8058	0.8058
0.7902	0.7902
0.817	0.817
0.8147	0.8147
0.7902	0.7902
0.8393	0.8393

0.85	0.85
0.85	0.85
0.8536	0.8536
0.875	0.875
0.8714	0.8714
0.8622	0.8622
0.8776	0.8776
0.8724	0.8724
0.8786	0.8786
0.8786	0.8786
0.8714	0.8714
0.8679	0.8679
0.8714	0.8714
0.8	0.8
0.7929	0.7929
0.7902	0.7902
0.75	0.75

Table B.3. Schedule small statement

ACO2	GA
0.8566	0.8566
0.8653	0.8653
0.7892	0.7892
0.7883	0.7883
0.8455	0.8455
0.9089	0.9089
0.8282	0.8282
0.7946	0.7946
0.8282	0.8282
0.8282	0.8282
0.8282	0.8282
0.8282	0.8282
0.8459	0.8459
0.9481	0.9567
0.8581	0.8581
0.8653	0.8653
0.844	0.844
0.844	0.844
0.844	0.844
0.8246	0.8246
0.8653	0.8653
0.8653	0.8653
0.8634	0.8634
0.8537	0.8537
0.8391	0.8391
0.8556	0.8556
0.8556	0.8556
0.8052	0.8052
0.8246	0.8246

0.8517	0.8517
0.8207	0.8207
0.9089	0.9089
0.7883	0.7883
0.9248	0.9248
0.7892	0.7892
0.9397	0.9397
0.9481	0.9481
0.7846	0.7846
0.799	0.799
0.82	0.82
0.8547	0.8547
0.8456	0.8456
0.803	0.803
0.8172	0.8172
0.8211	0.8211
0.8262	0.8262
0.8262	0.8262
0.7875	0.7875
0.8521	0.8521
0.8585	0.8585

2. Schedule2 Program

Table B.4 Schedule2 small block results

AS	GA
0.8585	0.8585
0.8623	0.8623
0.8623	0.8623
0.8623	0.8623
0.8019	0.8019
0.8019	0.8019
0.7151	0.7151
0.8434	0.8434
0.8569	0.8569
0.7594	0.7594
0.8129	0.8475
0.8129	0.7248
0.8073	0.7534
0.8852	0.8852
0.6545	0.6804
0.8423	0.8261
0.77	0.5743
0.6545	0.6804
0.816	0.6494
0.6929	0.7159
0.8145	0.8145
0.827	0.827

0.7241	0.6958
0.7759	0.7759
0.7792	0.7792
0.8082	0.8082
0.7995	0.7524
0.7406	0.7406
0.7406	0.7406
0.7406	0.7406
0.6887	0.6887
0.6887	0.6887
0.7406	0.7406
0.7406	0.7406
0.809	0.809
0.7151	0.7151
0.7943	0.7038
0.8245	0.7491
0.8623	0.8623
0.8472	0.8019
0.8467	0.8467
0.809	0.7335
0.7792	0.7491
0.7146	0.7146
0.8231	0.7759
0.7618	0.7618
0.8231	0.7759
0.7335	0.7335
0.7571	0.7571
0.7665	0.7665

Table B.5 Schedule2 small decision results

AS	GA
0.8851	0.8851
0.8829	0.8829
0.8874	0.8874
0.8739	0.8739
0.8851	0.8851
0.8829	0.8829
0.9356	0.9356
0.8135	0.8135
0.8547	0.8547
0.873	0.873
0.8162	0.8162
0.8784	0.8784
0.8432	0.8432
0.8284	0.8284
0.8514	0.8514
0.8581	0.8581
0.8338	0.8338
0.8604	0.8604
0.8622	0.8622

0.8378	0.8378
0.8162	0.8162
0.8453	0.8453
0.8529	0.8529
0.8694	0.8694
0.8604	0.8604
0.8529	0.8529
0.8559	0.8559
0.8168	0.8168
0.8604	0.8604
0.8814	0.8814
0.8453	0.8453
0.7988	0.7988
0.8405	0.8405
0.8904	0.8904
0.8716	0.8716
0.8446	0.8446
0.8159	0.8159
0.8168	0.8168
0.8438	0.8438
0.8733	0.8733
0.8497	0.8497
0.8784	0.8784
0.8834	0.8834
0.8328	0.8328
0.8834	0.8834
0.8818	0.8818
0.8829	0.8829
0.885	0.885
0.8589	0.8589
0.8904	0.8904

Table B.6 Schedule2 small statement

AS	GA
0.8597	0.8597
0.8465	0.6794
0.886	0.8811
0.8588	0.7
0.8574	0.7167
0.8654	0.8654
0.8693	0.713
0.8026	0.713
0.7976	0.737
0.7744	0.7078
0.8523	0.6646
0.7997	0.7397
0.7781	0.71
0.8293	0.7875
0.7875	0.7644

0.8228	0.8206
0.7745	0.7782
0.8051	0.6236
0.694	0.6273
0.8805	0.5986
0.8208	0.6375
0.8856	0.8856
0.8926	0.8926
0.882	0.882
0.8683	0.8683
0.8947	0.8947
0.8931	0.8931
0.8264	0.6755
0.8156	0.8156
0.6833	0.6833
0.8136	0.8136
0.8274	0.8111
0.8417	0.8417
0.7652	0.7489
0.7991	0.7991
0.8324	0.8324
0.7543	0.7543
0.7417	0.7417
0.6141	0.7059
0.8304	0.7059
0.8689	0.8689
0.8541	0.8541
0.8363	0.8096
0.8615	0.8615
0.8846	0.8846
0.7969	0.792
0.8051	0.6236
0.7438	0.7438
0.7438	0.882
0.7875	0.7644

3. Printtokens Program

Table B.7 Printtokens small block result

AS	GA
0.7605	0.7626
0.7807	0.7922
0.7414	0.7507
0.7414	0.7507
0.7738	0.7804
0.7864	0.7864
0.8108	0.8115

0.7593	0.7599
0.7804	0.791
0.7685	0.7765
0.7533	0.7652
0.7619	0.7705
0.7447	0.7487
0.7011	0.7149
0.8565	0.8669
0.7612	0.7626
0.7508	0.7582
0.7444	0.7529
0.7656	0.7757
0.7228	0.7307
0.7302	0.7503
0.7418	0.745
0.7677	0.7778
0.7656	0.7815
0.755	0.7603
0.7872	0.7902
0.7287	0.7401
0.7421	0.7535
0.7535	0.7664
0.7321	0.744
0.7634	0.7753
0.6956	0.7302
0.732	0.7521
0.7376	0.7572
0.7992	0.8008
0.7778	0.781
0.7518	0.7612
0.7518	0.8312
0.8283	0.8312
0.8492	0.8521
0.7547	0.7605
0.6938	0.7149
0.7447	0.7487
0.7626	0.7705
0.7857	0.7922
0.8115	0.8115
0.7751	0.7864
0.7593	0.7599
0.7414	0.7652
0.7685	0.7765
0.7811	0.791

Table B.8 Printtokens small decision results

AS	GA
0.8004	0.8101
0.7929	0.8014

0.7819	0.7957
0.7852	0.7988
0.7848	0.79929
0.8024	0.8033
0.8195	0.8204
0.7893	0.7975
0.885	0.885
0.8539	0.855
0.8241	0.8257
0.7458	0.7621
0.8243	0.8371
0.8577	0.8596
0.808	0.8152
0.8085	0.8348
0.8957	0.8957
0.8703	0.8703
0.8752	0.8752
0.8629	0.8636
0.875	0.875
0.8562	0.857
0.859	0.8596
0.8362	0.8362
0.8166	0.8166
0.8465	0.8465
0.8709	0.8709
0.8252	0.8252
0.8149	0.8236
0.8268	0.835
0.8263	0.8301
0.8252	0.8296
0.7979	0.8123
0.8008	0.8144
0.7688	0.7836
0.7918	0.7988
0.8189	0.8258
0.7933	0.7983
0.8189	0.8258
0.786	0.804
0.8041	0.8171
0.7802	0.797
0.7611	0.7755
0.7744	0.786
0.7645	0.774
0.8158	0.831
0.7865	0.8013
0.7901	0.8075
0.793	0.8087
0.804	0.8104

Table B.9 Printtokens small statement

AS	GA
0.8445	0.845
0.8907	0.8926
0.9045	0.9045
0.9064	0.907
0.8358	0.8488
0.8554	0.8558
0.8932	0.8963
0.9108	0.9112
0.8758	0.8767
0.8504	0.8548
0.8767	0.8776
0.8813	0.8846
0.8876	0.8905
0.8143	0.822
0.8732	0.8592
0.8592	0.822
0.8434	0.8543
0.8577	0.8625
0.8467	0.8471
0.8728	0.8791
0.9049	0.9071
0.8806	0.8949
0.8504	0.8526
0.884	0.886
0.8595	0.864
0.8767	0.8777
0.8573	0.8665
0.8604	0.8614
0.8879	0.8895
0.8949	0.8987
0.8164	0.8349
0.8464	0.8546
0.8175	0.8414
0.8476	0.8529
0.8543	0.8572
0.8574	0.872
0.8085	0.8276
0.824	0.8443
0.7884	0.8068
0.8701	0.88
0.8701	0.88
0.8358	0.8472
0.7808	0.8138
0.8127	0.8438
0.8158	0.8435
0.8105	0.8286
0.801	0.8228
0.7978	0.8138

0.8218	0.8316
0.8671	0.868

4. Printtokens2 Program

Table B.10 Printtokens2 small block results

AS	GA
0.8513	0.8398
0.8223	0.812
0.7979	0.8027
0.7883	0.8069
0.854	0.8451
0.769	0.7973
0.8265	0.8224
0.8515	0.8354
0.8475	0.8378
0.8142	0.8177
0.8305	0.824
0.8583	0.8283
0.8383	0.8015
0.8558	0.8267
0.8209	0.815
0.8366	0.8366
0.8402	0.8256
0.8213	0.8195
0.8366	0.8366
0.7102	0.7801
0.8363	0.8372
0.8517	0.8517
0.8471	0.8364
0.8517	0.8463
0.8174	0.8186
0.8635	0.8571
0.8452	0.8452
0.843	0.843
0.8329	0.8329
0.8731	0.8329
0.8112	0.8689
0.8245	0.8218
0.8598	0.855
0.8744	0.8744
8597	0.8578
0.8355	0.8096
0.8246	0.8125
0.823	0.8248
0.8016	0.7725
0.8135	0.8135
0.8762	0.8762
0.8329	0.8259

0.7899	0.776
0.8089	0.8142
0.8213	0.8222
0.8045	0.7948
0.8107	0.8107
0.8335	0.8044
0.8442	0.8469
0.8296	0.8286

Table B.11 Printtokens2 small decision

AS	GA
0.8321	0.8422
0.8156	0.8227
0.8298	0.838
0.8332	0.8411
0.8556	0.8613
0.8421	0.8527
0.8312	0.8407
0.8559	0.86
0.8758	0.8788
0.8535	0.8584
0.8596	0.8628
0.8585	0.8613
0.8225	0.8247
0.8655	0.8683
0.8622	0.8669
0.8557	0.8662
0.8333	0.842
0.8028	0.8101
0.8479	0.8553
0.8684	0.8753
0.8613	0.866
0.8683	0.872
0.8307	0.8363
0.8661	0.8746
0.8503	0.8522
0.8694	0.8747
0.8553	0.8567
0.8526	0.8566
0.8045	0.8195
0.8292	0.83
0.8297	0.8377
0.8641	0.8676
0.7971	0.8088
0.8611	0.8621
0.8237	0.8312
0.8721	0.8727
0.825	0.8421
0.8107	0.8298

0.8579	0.8579
0.8672	0.8672
0.8189	0.8189
0.8218	0.8283
0.829	0.829
0.8518	0.8529
0.8293	0.8369
0.8113	0.8267
0.8198	0.8336
0.8234	0.8329
0.8985	0.9002
0.8173	0.8173

Table B.12 Printtokens2 small statement

AS	GA
0.8598	0.8595
0.8538	0.8639
0.8188	0.8313
0.8776	0.8838
0.8574	0.8636
0.8563	0.8582
0.8516	0.8559
0.831	0.8416
0.8359	0.8401
0.886	0.8867
0.8691	0.8767
0.8418	0.8443
0.862	0.864
0.8792	0.882
0.8493	0.8531
0.8659	0.8751
0.859	0.8605
0.882	0.8848
0.8522	0.8581
0.8717	0.8737
0.8792	0.8811
0.8475	0.8475
0.8489	0.8526
0.8419	0.848
0.8519	0.8519
0.8681	0.8681
0.831	0.8393
0.8202	0.8247
0.827	0.8333
0.8641	0.8667
0.8384	0.8434
0.8093	0.8267
0.8418	0.8418
0.8364	0.8429

0.8514	0.8521
0.8276	0.8312
0.8455	0.8464
0.8586	0.8586
0.827	0.8354
0.8444	0.8508
0.8406	0.8428
0.8175	0.8249
0.809	0.827
0.8081	0.8182
0.7997	0.8092
0.8288	0.8345
0.8008	0.8058
0.8148	0.8154
0.8297	0.8355
0.838	0.851

Results of large matrices

1. Schedule Program

Table B.13 Schedule Large block results

AS	GA
0.983	0.9625
0.9857	0.9942
0.9876	0.9949
0.9848	0.9724
0.9948	0.9956
0.988	0.9938
0.9916	0.9707
0.9968	0.9968
0.9875	0.9805
0.9946	0.9816
0.9966	0.9966
0.989	0.9959
0.9584	0.9963
0.9975	0.9975
0.989	0.991
0.9915	0.9789
0.9977	0.9977
0.9939	0.9844
0.991	0.9962
0.998	0.998
0.9904	0.9967
0.9951	0.998
0.9943	0.987

0.9961	0.997
0.9977	0.9977

Table B.14 Schedule Large decision results

AS	GA
0.9973	0.9973
0.9943	0.9967
0.9968	0.9968
0.9948	0.9971
0.9954	0.9956
0.9938	0.9956
0.9963	0.9956
0.9958	0.9957
0.9965	0.9962
0.995	0.9975
0.9935	0.9953
0.9942	0.9966
0.994	0.9969
0.9915	0.9907
0.9969	0.9969
0.9942	0.9937
0.9945	0.9967
0.9947	0.9948
0.9957	0.9978
0.9961	0.997
0.9942	0.9943
0.9953	0.9972
0.9941	0.9941
0.9929	0.9949
0.996	0.9982

Table B.15 Schedule Large statement results

AS	GA
0.9945	0.995
0.9927	0.9971
0.9972	0.9972
0.9969	0.997
0.9971	0.9971
0.9936	0.9974
0.9957	0.9966
0.994	0.996
0.9985	0.9985
0.9963	0.9969
0.9965	0.9977
0.9984	0.9985
0.9983	0.9983
0.996	0.998
0.9945	0.9968

0.9967	0.9971
0.9966	0.9971
0.996	0.9975
0.9533	0.9963
0.9969	0.9969
0.9987	0.9987
0.9974	0.9974
0.9974	0.9974
0.9941	0.9952
0.9929	0.9951

2. Schedule2 Program

Table B.16 Schedule2 Large block results

AS	GA
0.9281	0.9281
0.934	0.934
0.8915	0.884
0.8019	0.8259
0.9282	0.7859
0.9572	0.9572
0.8975	0.8358
0.9691	0.8622
0.9637	0.9637
0.9758	0.9758
0.9777	0.9777
0.9781	0.91
0.9772	0.9781
0.9857	0.9857
0.9941	0.9941
0.9772	0.934
0.9825	0.928
0.9947	0.9947
0.9886	0.9886
0.9485	0.9489
0.9446	0.9503
0.8612	0.8477
0.98	0.9522
0.9684	0.8854
0.9629	0.9629

Table B.17 Schedule2 Large decision results

AS	GA
0.9916	0.9952
0.9958	0.9959
0.9888	0.9892

0.9913	0.9927
0.9962	0.9962
0.9803	0.9826
0.9958	0.9959
0.9981	0.9981
0.9958	0.9958
0.9961	0.9961
0.9963	0.9963
0.996	0.9964
0.9974	0.9975
0.9964	0.9966
0.9964	0.9968
0.9975	0.9976
0.9978	0.9978
0.9943	0.9942
0.9918	0.9924
0.9903	0.9942
0.9932	0.9963
0.9899	0.9904
0.9905	0.9898
0.9977	0.9977
0.997	0.997

Table B.18 Schedule2 Large statement results

AS	GA
0.9747	0.9756
0.885	0.8872
0.9558	0.9558
0.8471	0.8471
0.983	0.983
0.9963	0.9969
0.9631	0.9641
0.9913	0.9954
0.9869	0.9871
0.9749	0.9755
0.987	0.9878
0.9934	0.9974
0.9903	0.9905
0.9925	0.9927
0.9918	0.992
0.9903	0.9911
0.987	0.9882
0.9881	0.9914
0.9973	0.9942
0.987	0.9893
0.9829	0.9853
0.9573	0.9599
0.9925	0.9925
0.9903	0.9904
0.991	0.991

3. Printtokens Program

Table B.19 Printtokens Large block results

AS	GA
0.9844	0.9879
0.9842	0.9842
0.9892	0.9899
0.9905	0.9906
0.9881	0.9892
0.9874	0.9893
0.9892	0.9906
0.9803	0.9884
0.9815	0.9833
0.991	0.9918
0.9816	0.9833
0.993	0.994
0.9867	0.9926
0.984	0.9838
0.987	0.9898
0.9795	0.9833
0.986	0.9856
0.9899	0.9916
0.9899	0.9925
0.9915	0.9924
0.9908	0.9919
0.9927	0.9964
0.9834	0.8058
0.9898	0.99
0.9958	0.9956

Table B.20 Printtokens Large decision results

AS	GA
0.9801	0.9835
0.985	0.9878
0.9743	0.9831
0.9825	0.9843
0.9902	0.9918
0.9821	0.9857
0.994	0.9945
0.9866	0.9881
0.9805	0.9838
0.9927	0.9939
0.9789	0.9813
0.9878	0.9888
0.9841	0.9879

0.9854	0.987
0.986	0.987
0.9857	0.9863
0.99	0.9919
0.9854	0.9886
0.986	0.9885
0.9848	0.9846
0.9949	0.9952
0.9896	0.991
0.9861	0.9876
0.984	0.9871
0.9872	0.9876

Table B.21 Printtokens Large statement results

AS	GA
0.9921	0.9925
0.988	0.9865
0.9902	0.9914
0.9828	0.9865
0.9843	0.9857
0.9934	0.9938
0.989	0.9903
0.9884	0.9899
0.9819	0.986
0.9896	0.9897
0.9876	0.9893
0.987	0.9881
0.9849	0.9882
0.9911	0.9942
0.9942	0.9947
0.9954	0.9956
0.9886	0.9925
0.9941	0.9943
0.9918	0.9943
0.9911	0.9936
0.9898	0.9908
0.9927	0.995
0.9923	0.9948
0.995	0.9953
0.9929	0.9953

4. Printtokens2 Program

Table B.22 Printtokens2 Large block result

AS	GA
0.988	0.9854
0.9911	0.9653
0.9933	0.9803
0.9892	0.9806
0.9938	0.9799
0.9923	0.9861
0.9931	0.991
0.9931	0.9836
0.9895	0.9866
0.9901	0.9855
0.9956	0.9887
0.9918	0.9939
0.9952	0.9845
0.987	0.9777
0.9897	0.9788
0.9951	0.9915
0.9955	0.9907
0.9881	0.9816
0.9949	0.9796
0.9855	0.9771
0.9904	0.9749
0.9776	0.9675
0.9732	0.9752
0.98	0.9877
0.9894	0.9898

Table B.23 Printtokens2 Large decision result

AS	GA
0.9834	0.9862
0.9914	0.9926
0.9956	0.9959
0.9957	0.9953
0.9938	0.9942
0.9961	0.9964
0.9967	0.9968
0.9922	0.9036
0.9939	0.9943
0.9906	0.9929
0.9914	0.9927
0.9912	0.9927
0.9914	0.9935
0.9942	0.9924
0.9939	0.9959

0.9953	0.9958
0.9947	0.9962
0.9949	0.9953
0.9941	0.9946
0.9932	0.9934
0.9946	0.9945
0.9943	0.9937
0.9809	0.9867
0.9915	0.9931
0.9936	0.9946

Table B.24 Printtokens2 Large statement result

AS	GA
0.9907	0.992
0.9909	0.9932
0.9943	0.9925
0.9937	0.9939
0.9907	0.9925
0.991	0.9914
0.9944	0.9943
0.9941	0.9938
0.992	0.9944
0.9954	0.9935
0.9955	0.9955
0.9944	0.9928
0.9935	0.9942
0.9902	0.9924
0.994	0.9939
0.9928	0.9951
0.9951	0.9949
0.9943	0.9944
0.9921	0.9924
0.9961	0.9957
0.9878	0.9899
0.9924	0.9916
0.996	0.9952
0.9959	0.9958
0.9964	0.9962

حالات الفحص حسب الاولوية استخدام خوارزمية مملكة النمل لترتيب

اعداد

سميه صابر أحمد اسماعيل

المشرف

الدكتور خليل الهندي

ملخص

ان عملية فحص البرمجيات هي خطوة مهمة في دورة حياة البرمجيات و عند تطويرها, حيث أنه تستخدم للتحقق من صحة و كمالية و أمن و جودة برمجيات الحاسوب و تطبيقاته.

ان فحص البرمجيات تتضمن عملية تنفيذ البرنامج و تطبيقه مع الرغية في ايجاد الاخطاء و معرفة الفروقات بين البرنامج الحالي و البرنامج المطلوب من سلوك و من وظائف. عملية فحص البرمجيات تبدأ مع بداية تطوير البرمجيات و هذا يعني أن فحص البرمجيات تلزم في عملية تطوير البرنامج بأكملها.

ان اختبار الانحدار هي عملية تنفيذ صيانة متكرر للتأكد و التحقق من صحة البرنامج المعدل و المطور. متى ما تم تعديل برنامج أو تطور, تطرأ الحاجة لايجاد حالات فحص جديدة للبرنامج لفحص التطورات وو المستجدات التي حدثت للبرنامج و تضاف هذه الحالات لمجموعة حالات الاختبار مما يؤدي الى زيادة حجم حالات الاختبار بشكل كبير مما يحتاج الى و يتطلب لوقت و جهد كبيرين لعملية اختبار الانحدار.

فعلى سبيل المثال ان عملية فحص برنامج مكون من 20000 سطر من الرموز يحتاج الى 7 أسابيع من الوقت لتنفيذ مجموعة حالات الفحص. لهذه الاسباب فأن الباحثين أوجدوا عدة طرق تقلل من الوقت و الجهد اللازم لعملية اختبار الانحدار ومنها, اختيار حالات الفحص و الاختبار, تقليل حالات الفحص, و ترتيب حالات الفحص.

ان عملية اختيار حالات الفحص تقوم باختيار جزء من حالات الفحص و تنفيذها اعتمادا على معلومات من البرنامج القديم و الجديد. أما عملية تقليل حالات الفحص يقلل من حجم مجموعة حالات الفحص الى ؟أصغر حجم مستطاع يغطي جميع حالات الفحص الاصلية بوجود شرط محدد.

أما عملية ترتيب حالات الفحص تقوم بترتيب حالات الفحص لتقوم بفحص الرنامج ضمن معيار معين كفحصه للوصول الى فحص كامل للرموز في أقرب وقت ممكن أو للوصول الى أكبر عدد من الاخطاء في أقرب وقت ممكن.

أن التقنيات المستخدمة لترتيب حالات الفحص تزيد من كفاءة و أداء اختبار الانحدار حيث أنها لا تقوم بتضييع أي حالة من حالات الفحص بترتيبها حسب معيار معين. لذا فأن أيجاد ترتيب فعال و ناجع لحالات الفحص يعتبر مشكلة صعبة بحاجة لوقت طويل , لذا فأن اجاد ترتيب لحالات الفحص يجب أن يكون اجتهادي.

ان خوارزمية مملكة النمل تعتبر خوارزمية ناجعة لايجاد حلول لمشاكل صعبة حيث أنها خوارزمية اجتهادية استخدمت لحل كثير من المشاكل كمشكلة رجل المبيعات. و لان هذه الخوارزمية لم تستخدم بعد لترتيب حالات الفحص في اختبار الانحدار فأن هذه الرسالة ستقوم بأيجاد حل عن طريق استخدام خوارزمية مملكة النمل لايجاد ترتيب ناجع لحالات الفحص حسب الاولوية. هذه الرسالة ستقوم بعمل مقارنة بيا الخوارزمية المقترحة و بين خوارزمية الجينية و ايجاد الفروق بينهما.